Universidade de Santiago de Compostela
Departamento de Electrónica e Computación

MEJORA DE LOS ALGORITMOS DE LATENCIA VARIABLE PARA EL CÁLCULO DE LA DIVISIÓN, LA RAÍZ CUADRADA, Y SUS RECÍPROCOS

Presentada por:
Daniel Piso Fernández

Dirigida por:
Javier Díaz Bruguera

Santiago de Compostela, Julio de 2009

Dr. **Javier Díaz Bruguera**, Profesor Catedrático de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

HACE CONSTAR:

Que la memoria titulada **Mejora de los algoritmos de latencia variable para el cálculo de la división, la raíz cuadrada, y sus recíprocos** ha sido realizada por D. **Daniel Piso Fernández** bajo nuestra dirección en el Departamento de Electrónica e Computación de la Universidade de Santiago de Compostela, y constituye la Tesis que presenta para obtar al grado de Doctor en Ciencias Físicas.

Santiago de Compostela, Julio de 2009

Asdo: **Javier Díaz Bruguera**
Director da tese de licenciatura

Asdo: **Javier Díaz Bruguera**
Director del Departamento de Electrónica y Computación

Asdo: **Daniel Piso Fernández**
Autor de la Tesis

Agradecimientos

Una tesis es un trabajo muy largo en el tiempo. Y en tanto tiempo pasan muchas cosas que cuando se llega al final es inevitable recordar. Hay momentos malos y momentos buenos, pero solo hay que acordarse de lo positivo. Y en lo positivo siempre está la gente que a uno le rodea, que contribuye tanto o más que uno mismo. Gracias a todos.

A mi director de tesis D. Javier Díaz Bruguera, gracias por su dirección científica y su impagable apoyo a lo largo de todos estos años.

He tenido muchos apoyos dentro del Departamento. Gracias a Montse, Roberto y Elisardo. Muchas gracias por sus consejos y sus continuas palabras tranquilizadoras. Han sido de mucha ayuda. A los compañeros del Departamento de Electrónica y Computación. En especial a Diego y a Xulio.

Este trabajo ha sido desarrollado gracias a la financiancion aportada por el Gobierno de España y la Xunta de Galicia a traves de los proyectos TIN2004-07797-C02-01, TIN2007-67537-C03-01 y PGIDIT06TI2060PR.

A Patricia, por conseguir que me levante por las mañanas, porque algunas cosas ahora tengan más sentido del que tenían, por poner un poco de alegría al asunto..., en fin, todas esas cosas que tanto cuesta expresar...

A mis padres Manuel y María Inés. Esperar a que un hijo acabe esto de la tesis y se dedique a un trabajo tan raro sólo denota un apayo incondicional e irrenunciable. Y eso hay que valorarlo porque no todo el mundo lo tiene. A mi hermana Carmela, por su confianza ciega en mí. Gracias a mi tía Ofelia por haber crecido a su ritmo con nosotros. A mi abuelo Daniel por enseñarme cosas que no están en los libros. Gracias a Manolo, Marisa y Carmen por acogerme desde el primer día. Y por valorarme más de lo que me merezco. No hay nada más grande en el mundo que encontrarse con buenas personas.

Desafortunadamente, solemos ser incapaces de admirar en su justa medida a las personas hasta que llega su ausencia. Y de esas hemos tenido muchas últimamente. Todas ellas eran auténticas fuerzas de la naturaleza. A Preciosa, al tío Manuel, a Agustín, a Divina, a Pancho, a la abuela Carmen. Cuando uno no sabe que hacer solo tiene que acordarse de lo que ellos harían.

Gracias a mis tíos Antonio y María José por ser una referencia para mi. A mis primos Javier, David y Ana porque hemos crecido juntos. A mis padrinos Carmen y Daniel por el gran cariño que me tienen. Y a todos los demás, que ya no caben.

Gracias a los amigos, los que lo son lo saben. Gracias a los que han sido, a los que estuvieron, a los que no están y a los que vendrán.

Julio de 2009

Tu destino está en los demás
tu futuro es tu propia vida
tu dignidad es la de todos

José Agustín Goytisolo

Índice general

Índice de figuras

Índice de tablas

Introducción

En las dos décadas pasadas se han hecho múltiples avances en el diseño de microprocesadores. Estos avances han hecho posible la construcción de aplicaciones que resuelven problemas más complejos. Estas aplicaciones demandan cada vez más velocidad y requerimientos de rendimiento más severos. Por estas razones, existe una fuerte demanda de cálculos en punto flotante de alto rendimiento. La división, el recíproco, la raíz cuadrada y la raíz cuadrada recíproca en punto flotante son funciones matemáticas muy importantes en este tipo de aplicaciones. Un calculo lo más rápido y exacto posible de estas operaciones matemáticas se ha vuelto irrenunciable. Se ha demostrado que el cómputo eficiente de estas funciones produce mejoras muy notables en el rendimiento de los procesadores [OF96].

Más aún, estas funciones se están volviendo, cada vez, más importantes en distintas aplicaciones en diferentes áreas. Unidades de procesamiento de gráficos (GPU's), procesadores digitales de señal (DSP's), unidades de punto flotante (FPU's), unidades de procesamiento físico (PPU's) o circuitos de aplicación específica (ASIC's) (ver, por ejemplo, [Bla06, CS05, GWSH33, IHE$^+$00, Par99]) son ejemplos de aplicaciones hardware en diferentes campos que han producido implementaciones de alto rendimiento de estas funciones.

Existen dos tipos de algoritmos iterativos que pueden ser usados para computar la división, el recíproco, la raíz cuadrada y la raíz cuadrada recíproca en punto flotante. Estos algoritmos son válidos tanto para formatos de precisión simple o precisión doble del estándar IEEE 754[IEE08]. Por una parte, los algoritmos sustractivos o de dígito-recurrencia [EL94b] suelen producir unidades de pequeño tamaño pero su convergencia lineal hace que las latencias sean elevadas. Por otra parte, los algoritmos multiplicativos tienen convergencia cuadrática lo que produce una convergencia más rápida con un consumo considerable de área.

Los algoritmos sustractivos, debido a su convergencia lineal obtienen un número fijo de bits en cada una de sus operaciones. Existen soluciones, llama-

das de radix alto, que obtienen un número grande de bits en cada iteración. Este tipo de soluciones tienen latencias menores, a costa de incremento significativo en área. Aunque los dos tipos de algoritmo tienen sus ventajas y desventajas, cada vez, un mayor número de procesadores comerciales implementan algoritmos multiplicativos para realizar el cálculo de la división, la raíz cuadrada y sus recíprocos [Obe99, NDJD01]. Por esta razón este trabajo se dedica a los algoritmos multiplicativos.

Los dos algoritmos multiplicativos más conocidos son: el algoritmo de Newton-Raphson (NR) [WF82] y el algoritmo de Goldschmidt (GLD) [Gol64]. Tanto el algoritmo GLD como el algoritmo NR, como algoritmos multiplicativos que son, usan una aproximación inicial al resultado de baja precisión (también llamada *semilla*). Este valor es obtenido, casi siempre, utilizando un método basado en tablas [Mat01, PB02]. Esta aproximación inicial se usa para realizar un número determinado de iteraciones para obtener el resultado con la exactitud requerida. El número de iteraciones depende de la exactitud de la semilla y la exactitud requerida para el resultado. Ambos algoritmos realizan dos multiplicaciones por iteración el caso de la división y el recíproco. Si se calcula la raíz cuadrada o la raíz cuadrada recíproca se realizan tres multiplicaciones por iteración. Una de las características del algoritmo GLD es que estas multiplicaciones son independientes entre sí. De este modo estas multiplicaciones pueden ser realizadas, bien en paralelo o bien reusando un multiplicador segmentado. Esto hace que el algoritmo GLD sea especialmente adecuado para su implementación en hardware. Esta propiedad no está presente en el caso del algoritmo NR de modo que su estructura se hace más adecuada para ser implementada en software.

Otra diferencia importante entre ambos algoritmos, es que el algoritmo GLD no tiene la propiedad de auto-corrección, mientras que el algoritmo NR si la tiene. Esto significa que, en el caso del algoritmo GLD, las inexactitudes introducidas en las operaciones intermedias de su cálculo se acumulan y producen desviaciones en el resultado. Esto hace más difícil el seguimiento de los errores acumulados y propagados en las operaciones intermedias de un modo formal [ES03a]. Si no se tiene en cuenta el error de las operaciones intermedias, es decir, en una implementación con precisión infinita, el resultado de estos algoritmos es siempre menor que el resultado exacto. En otros palabras convergen 'por abajo'. Sin embargo, en las implementaciones hardware, al introducir error en las operaciones intermedias, el resultado puede ser mayor que el resultado exacto en algunos casos. De este modo el proceso de redondeo se complica [OF97].

El estándar IEEE 754 [IEE08] requiere que los resultados del recíproco, la división, la raíz cuadrada y la raíz cuadrada recíproca sean redondeados de acuerdo con las estrategias allí descritas. El problema es que los algoritmos multiplica-

tivos no producen de manera directa un resultado correctamente redondeado. Además, tampoco producen directamente el valor del resto de la operación que tiene que ser computado a mayores si se requiere. Existen dos estrategias para obtener un resultado correctamente redondeado de un algoritmo multiplicativo. La primera de ellas consiste en calcular el resultado con mucha más precisión y exactitud de la requerida [Mar90, IM99]. La otra posibilidad es usar el resto para realizar una corrección del resultado [OF97, Sch95]. La primera alternativa requiere realizar iteraciones adicionales del algoritmo y usar un tamaño de palabra mayor para las operaciones intermedias[1]. De esto modo la latencia y el área de estos diseños es mucho mayor. Por esta razón se prefiere la alternativa de la obtención del resto, que será la contemplada aquí. La manera tradicional de hacerlo es sumar una constante al resultado obtenido para luego redondearlo de acuerdo con el valor del resto y del bit de redondeo. El tiempo que se dedica a este proceso se añade a la latencia total del algoritmo. Este tiempo adicional es uno de los principales lastres que tiene el rendimiento de los algoritmos multiplicativos. Existen diferentes métodos que intentan reducir el tiempo dedicado a la etapa de redondeo [ES03b, Sch95] pero ninguno de ellos elimina completamente la multiplicación necesaria para el cálculo del resto. Por eso, esta tesis está dedicada al desarrollo de algoritmos de latencia variable para algoritmos multiplicativos que mejoren la eficiencia de los métodos ya existentes.

En este trabajo se proponen diversos métodos de redondeo de latencia variable. El primer capítulo está dedicado a la explicación de los fundamentos teóricos necesarios sobre los que se basan el redondeo. Se describen los distintos formatos de representación de números en punto flotante que propone el estándar IEEE 754. A continuación se realiza un repaso por los distintos tipos de algoritmos para el cálculo de la división, el recíproco, la raíz cuadrada y la raíz cuadrada recíproca. Se hablará de algoritmos sustractivos y de algoritmos multiplicativos. Se describirán con detalle los dos algoritmos multiplicativos más importantes: el algoritmo GLD y el algoritmo NR. Por último, se dedica una sección a explicar el método de redondeo tradicional para los algoritmos multiplicativos.

En el segundo capítulo se presenta un método para el diseño de unidades para el cálculo de división, recíproco, raíz cuadrada y raíz cuadrada recíproca basadas en el algoritmo GLD (Se presentará el mismo análisis para el algoritmo NR en un apéndice). Este método permite obtener el tamaño de palabra para las operaciones intermedias. Este tamaño fijará también el tamaño del multi-

[1] Concretamente, en [IM99] se demuestra que las cadenas de ceros (o unos) de la representación exacta en binario de un cociente no serán mas largas que la precisión objetivo. Entonces, para que el redondeo no afecte el resultado ha de ser calculado al menos con el doble de la precisión requerida

plicador. El método consiste en un análisis preciso del error del resultado que contabiliza las distintas contribuciones a dicho error. Estas contribuciones son, por un lado, el error del propio algoritmo (error de aproximación) y, por otro, el error de las operaciones intermedias (error del hardware). De este análisis se obtiene una expresión analítica del error del resultado para una iteración cualquiera. A través de esta expresión se calcularán los límites para los valores del error en cada iteración. Estos límites permiten obtener la longitud óptima para las operaciones intermedias, que fijan también el tamaño del multiplicador.

En el siguiente capítulo se describe una modificación de los algoritmos de redondeo ya existentes para conseguir una reducción del número de casos en los que el cálculo del resto es necesario. Como se había dicho, el redondeo típico de estos algoritmos consiste en transformar el resultado que se obtiene del algoritmo mediante la suma de una constante. El método tradicional consigue no tener que calcular el resto para algunos valores de los bits de redondeo. Para los demás casos es necesario calcular el resto. En función del valor del resto, si es necesario, y de los bits de redondeo se determina la acción de redondeo. La modificación propuesta utiliza información adicional del resultado antes de la transformación para reducir, aún más, el número de casos en los que el cálculo del resto es necesario.

El cuarto capítulo está dedicado a la descripción de un nuevo método de redondeo basado en la obtención en paralelo del resto. Se propone un nuevo método de obtención del resto en paralelo con la ejecución del algoritmo. Se obtiene a partir del resto de la iteración anterior y utiliza como operación básica la multiplicación. Este método produce un valor adecuado del resto en la gran mayoría de los casos. El resto obtenido es el que corresponde al resultado directamente obtenido del algoritmo sin ninguna transformación. Esto implica el diseño de un nuevo método de redondeo basado en este resto. Para aquellos casos en que el valor del resto obtenido no es el adecuado será necesario obtener el resto de manera convencional. Una vez que se tiene el valor del resto se aplica el nuevo método de redondeo para obtener el resultado redondeado adecuado.

En el último de los capítulos se presenta un método similar al método presentado en el capítulo anterior pero con la importante ventaja de no necesitar la operación de multiplicación. Dicho método también obtiene un valor del resto en paralelo con la ejecución del algoritmo. Dicho valor es adecuado en la mayoría de las ocasiones. En las que no lo es, también se hace necesario calcular el resto de modo tradicional. La diferencia está en que el cálculo del resto es mucho más sencillo que en el caso anterior. Basta con realizar una resta para obtener el valor del resto paralelo. Este método proporciona resultados muy parecidos al método anterior. Sin embargo, evita la utilización de un multiplicador adicio-

nal para el cálculo del resto. Esto supone un ahorro en recursos hardware muy importante.

Finalmente, por completitud, se presenta el análisis del error para el algoritmo de Newton-Raphson que se había explicado para el algoritmo de Goldschmidt en el segundo capítulo. Dado que el procedimiento es completamente análogo se presenta en un apéndice para no complicar la comprensión del método en dicho capítulo.

Capítulo 1

Fundamentos

En este capítulo se describen los fundamentos teóricos básicos para entender la computación del recíproco, la división, la raíz cuadrada inversa y la raíz cuadrada para punto flotante. Se abordan los distintos algoritmos de uso más general en la literatura para el cálculo de dichas funciones. Antes de explicar los algoritmos propiamente dichos se hace una descripción de los requerimientos que el estándar IEEE 754 [IEE08] impone a su cálculo. Por un lado, todo lo referente al formato de representación de los datos y, por otro, el formato que ha de tener el resultado. Este último aspecto es de gran importancia puesto que incluye la descripción del redondeo de los resultados, aspecto central de este trabajo.

1.1. Formatos de representación de números en punto flotante

Todos los sistemas de representación de números en punto flotante utilizan cuatro magnitudes diferentes para representar un número M_x [EL94a, Gol91]. Cada número estará compuesto por un bit de signo s_x, un exponente E_x, una mantisa normalizada X y una base β. El valor de éste será:

$$M_x = (-1)^{s_x} \cdot X \cdot \beta^{E_x} \tag{1.1}$$

El estándar IEEE 754 [IEE08] es el aceptado para la representación de los números en punto flotante en la aritmética de computadores. Propone un sis-

Parámetro	**Formato**		
	simple*	**doble***	**extendida***
bits totales	32	64	128
bits mantisa	24	53	113
bits exponente	8	11	15
exponente máximo	127	1023	16383
exponente mínimo	-126	-1022	-16382
polarización exponente	127	1023	16383

*El estándar denomina a los formatos de precisión simple, doble y extendida como *binary32*, *binary64* y *binary128* respectivamente

Tabla 1.1: Formatos del estándar IEEE 754 para punto flotante

tema de representación de números en punto flotante basado en tres formatos binarios y dos para representación decimal. El formato binario o decimal está especificado por el *radix*, es decir, la base. El formato binario está caracterizado por $\beta = 2$ y el decimal por por $\beta = 10$. Este apartado se ceñirá al caso binario que será el que nos ocupe. Para esta base la representación toma la siguiente forma:

$$M_x = (-1)^{s_x} \cdot X \cdot 2^{E_x} \tag{1.2}$$

El bit de signo tomará los valores $s_x = \pm 1$.

Una vez fijada la base, se proponen tres formatos básicos distintos dependiendo de la longitud del exponente y la mantisa. Para facilitar la representación y comparación los exponentes se representan en una notación polarizada que se explicará más adelante.

Los dos formatos básicos serán la *precisión simple*, *precisión doble* y el *formato extendido*.

Las longitudes del exponente y la mantisa para la representación de cada uno de los formatos se muestran en la Tabla 1.1. La mantisa siempre está normalizada, de modo que su primer bit siempre es 1 y no necesita ser almacenado. Este bit se denomina *bit oculto*. Así sólo es necesario almacenar 23, 52 o 112 bits dependiendo del formato (Figura 1.1). Este hecho sólo tiene una excepción, que son los *números denormales* o *subnormales*. El estándar los define como aquellos números en punto flotante, distintos de 0, cuya magnitud es menor que la más pequeña representable en el formato correspondiente. Estos no utilizan toda la precisión disponible para los números convencionales en el mismo formato. Su primer bit implícito es un 0

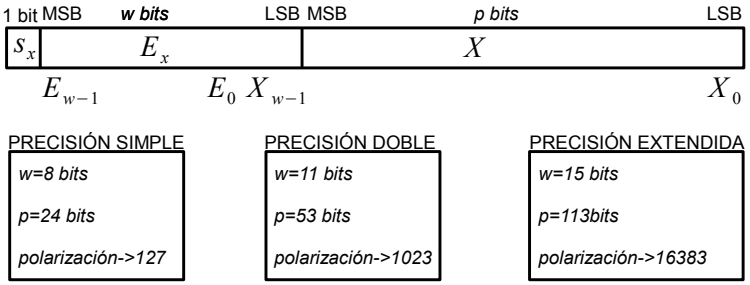


Figura 1.1: Representación del formato en punto flotante

En cuanto al exponente, se utiliza una representación polarizada. Así para obtener el valor del exponente hay que restar el valor correspondiente de la polarización (se pueden ver en la tabla 1.1 el valor de la polarización y los valores máximos y mínimos) , dependiendo del formato de la precisión. Se reservan dos valores para representar valores especiales. De acuerdo con el estándar, además del cero y los números con signo, se requiere que el estándar proporcione dos infinitos ($+\infty$ y $-\infty$) y dos *NaN*. Un *NaN* (*not a number*) es un dato simbólico en punto flotante. Existen dos tipos de representaciones *NaN*: *quiet* (*qNaN*) and *signalling* (*sNaN*). Los *qNaN* se caracterizan porque no llevan asociada ninguna propagación de señales de excepción, mientras que los *sNaN* propagan una excepción de operación inválida cuando son operandos.

Los valores especiales corresponden al valor mínimo del exponente E_{min}, que se usa para $+0$, -0 y *números denormalizados*, y al valor máximo del exponente E_{max}, reservado para $+\infty$, $-\infty$ y $NaNs$. Así de la combinación de los valores de los distintos campos del número en punto flotante surgen los siguientes números especiales:

- E_{max} y una mantisa distinta de cero representa un NaN. El valor de la mantisa dependerá de la máquina en concreto permitiéndose distintos $NaNs$.
- E_{max} y una mantisa cero representa $+\infty$ o $-\infty$ dependiendo del valor del bit de signo.
- E_{min} y una mantisa distinta de cero representa un número denormalizado.

Parámetro	Formato extendido asociado con:		
	simple	**doble**	**extendida**
bits totales	$\geq$32	$\geq$64	$\geq$128
exponente máximo	$\geq$1023	$\geq$16383	$\geq$65535

Tabla 1.2: Parámetros para los formatos extendidos IEEE 754 para punto flotante

- E_{min} y una mantisa igual a cero representa $+0$ o -0 dependiendo del valor del bit de signo.

Los usos de los números especiales se describen a continuación. El resultado del desbordamiento superior de una operación o de la división de un número real por 0 es $+\infty$ o $-\infty$ dependiendo del valor del bit de signo del resultado intermedio o del dividendo. El resultado de un desbordamiento inferior es $+0$ o -0 dependiendo del valor del bit de signo del resultado intermedio. Los números denormalizados son usados para el *desbordamiento inferior gradual* para representar valores próximos a los mínimos representables y no producir errores en determinadas operaciones aritméticas.

Finalmente, el estándar recoge también formatos de precisión *extendidos* y *extensibles*. Un formato de precisión extendido es un formato, que a partir de un formato básico soportado, incrementa su precisión y el rango de representación de números. Los extensibles son aquellos que están definidos por el usuario. Estos formatos se definen en base a una precisión y un rango para el exponente mínimos (el rango de exponente mínimo el igual al del siguiente formato básico más ancho). Estos valores están especificados en la tabla 1.2.

1.2. Algoritmos

Existen diversos algoritmos para el cálculo del recíproco, la división, la raíz cuadrada recíproca y la raíz cuadrada. Cada una de ellos con distinta complejidad y rendimiento. La naturaleza relativamente compleja e iterativa de estas funciones hacen que su cálculo sea sensiblemente más complicado que el de operaciones elementales como la suma y la multiplicación. Existen en la literatura científica actual dos grandes grupos de algoritmos para este propósito: los algoritmos sustractivos (*digit-recurrence*)[EL94a] y los algoritmos multiplicativos o de iteración funcional [EL94b]

1.2.1. Algoritmos sustractivos

Este tipo de métodos para la aproximación de funciones en hardware son conocidos como algoritmos *digit-recurrence* (DR). En estos algoritmos y sus implementaciones el resultado se representa en un *radix*, es decir, una base r. Se obtiene un dígito por cada iteración de manera que tienen una convergencia lineal. De este modo en cada iteración de estos algoritmos se obtiene un número fijo de bits. Este tipo de algoritmos son aplicables al recíproco, la división, la raíz cuadrada recíproca y la raíz cuadrada. También son válidos para otras funciones como el logaritmo y la exponencial y otras funciones[EL94a, PEB03, PB03]. Sus implementaciones son típicamente de baja complejidad, de área pequeña pero a cambio acostumbran a tener latencias altas.

La implementación de estos algoritmos puede ser combinacional o secuencial y pueden estar o no segmentados. El espacio de diseño es muy grande porque implica un gran número de parámetros y la mejor solución depende de los requerimientos particulares de cada caso. No es efectivo describir un conjunto de diseños y se opta por dar una explicación de algoritmos DR basada en ejemplos concretos de implementación [EL94b, DHH97] que son representativos.

Se presentarán aquí los algoritmos para operandos y resultado fraccionales y normalizados, que es la implementación adecuada para la representación de números en formato de punto flotante [EL94b]. Esta explicación se concentra en algoritmos que usan *conjuntos de dígitos redundantes* (que serán explicados a continuación) que tienen ventajas significativas en coste y velocidad. Los algoritmos presentados aquí son del tipo *MSDF-(most significant digit first)* en los que los diferentes dígitos del resultado son calculados empezando por el más significativo. El problema más complejo en la implementación de los algoritmos DR es la selección de los dígitos del resultado

A. Conjuntos de dígitos redundantes

Para acelerar el cálculo del resultado se usan los conjuntos de dígitos reduntantes. El conjunto estándar de dígitos para radix r es $\{0, 1, ..., r-1\}$. En un conjunto de dígitos redundantes el número de ellos es mayor que r. Para la representación del cociente la mayoría de ellos son de la forma

$$q_j \ \in \ D_a = \{-a, -a+1, ..., -1, 0, 1, ..., a-1, a\} \tag{1.3}$$

Esto es, conjuntos simétricos de enteros consecutivos con dígito máximo a. Para que D_a sea calificado como redundante, a debe satisfacer la relación

$a \geq \lceil \frac{r}{2} \rceil$. El grado de la redundancia se mide como el factor de redundancia, definido como

$$\rho = \frac{a}{r-1},\ \rho > \frac{1}{2} \tag{1.4}$$

Una representación con $a = \lceil \frac{r}{2} \rceil$ recibe el nombre de mínimamente redundante, mientras que con $a = r-1$ ($\rho = 1$) se llama máximamente redundante. Si $a > r-1$ ($\rho > 1$) es conocida como sobreredundante. Cualquier representación donde $a = \frac{r-1}{2}$ es no redundante.

B. División y recíproco

Los algoritmos DR consisten en n iteraciones de una recurrencia en la que cada iteración se obtiene un dígito del resultado. Sea $q[j]$ el valor del cociente despues de j iteraciones:

$$q[j] = \sum_{i=1}^{j} q_i \cdot r^{-i} \tag{1.5}$$

El cociente final de n dígitos será

$$q = \sum_{i=1}^{n} q_i \cdot r^{-i} \tag{1.6}$$

Supongamos una división caracterizada por un dividendo x y un divisor d.El algoritmo debe de producir un cociente q con un error menor de $1ulp$. Luego para obtener el cociente de manera adecuada dígito a dígito, despúes de la iteración j, el error ($\varepsilon[j]$) debe estar limitado por:

$$\varepsilon[j] = |\frac{x}{d} - q[j]| < r^{-j} \tag{1.7}$$

De esta ecuación se obtiene la siguiente desigualdad.

$$|x - d \cdot q[j]| < dr^{-j} \tag{1.8}$$

Se define, entonces, el resto parcial para la iteración j.

$$\omega[j] = r^j \cdot (x - d \cdot q[j]) \tag{1.9}$$

A partir de esta ecuación se obtiene una recurrencia que es la que se utiliza para la obtención del resultado.

$$\omega[j+1] = r \cdot \omega[j] - d \cdot q_{j+1} \tag{1.10}$$

La ecuación define la recurrencia de la división donde q_{j+1} es el $j+1$-ésimo dígito del cociente, numerados desde el bit de orden más alto al de orden más bajo. El valor $\omega[j]$ es el resto parcial en la iteración j. Inicialmente, $\omega[0] = x$, es decir, el primer resto parcial se fija al valor del dividendo.

El requerimiento fijado en (1.8) se traduce en la siguiente desigualdad que ha de cumplirse en todas las iteraciones

$$-d < \omega[j] < d \tag{1.11}$$

El dígito del cociente en la ecuación (1.10) deberá ser escogido de manera que $\omega[j+1]$ cumpla la desigualdad anterior. Esta labor la realiza la función de selección, que se describe a continuación.

C. Selección de dígitos del cociente

La selección eficiente de los dígitos del cociente es un problema que durante largo tiempo ha sido una barrera para la implementación eficiente de estos algoritmos. Los algoritmos DR utilizan métodos sofisticados y eficientes gracias a las representación de dígitos redundantes [EL94b]. En la ecuación (1.10) la selección del cociente está limitada por unos valores que ahora definimos como:

$$\forall j \ \underline{B} \leq \omega[j] \leq \bar{B} \tag{1.12}$$

Se puede demostrar que $\underline{B} = -\rho \cdot d$ y que $\bar{B} = \rho \cdot d$, donde ρ es el grado de redundancia del conjunto de dígitos D_a. Supongamos un intervalo de selección de $r \cdot \omega[j]$ para $q_{j+1} = k$ que está definido como $[L_k, U_k]$, esto es , el rango de valores para los cuales $\omega[j+1] = r \cdot \omega[j] - d \cdot k$ está correctamente acotado. Se puede demostrar que

$$U_k = (\rho + k) \cdot d \quad y \quad L_k = (-\rho + k) \cdot d \tag{1.13}$$

para una selección de dígitos correcta. Es condición se llama *de contención* y debe de cumplirse para cualquier función de selección de dígitos del cociente. El otro prerrequisito para una selección correcta se conoce como condición *de continuidad*. Para que se cumpla cualquier valor de $r \cdot \omega[j]$ debe de estar en algún intervalo de selección. Este hecho se puede expresar como

$$U_{k-1} \geq L_k \tag{1.14}$$

Los límites para intervalos sucesivos deben de coincidir o solaparse.

La recurrencia para la división, los límites para el resto parcial, y el intervalo de selección puede ser representado en un diagrama de Roberts [EL94a] (figura 1.2a). Este diagrama tiene como ejes el resto parcial desplazado $r\omega[j]$ y el resto parcial de la siguiente iteración $\omega[j+1]$. Representa la recurrencia mediante líneas para el parámetro k con valores $k = -a...a$. También se representan los límites para el resto parcial a través del rectángulo $\omega[j+1] = \rho d$, $\omega[j+1] = -\rho d$, $r\omega[j+1] = r\rho d$ y $r\omega[j+1] = -r\rho d$. El intervalo de selección para $q_{j+1} = k$ se obtiene de la proyección de la línea correspondiente sobre el eje $r\omega[j+1]$.

Otro diagrama que es útil en el diseño de funciones de selección es el diagrama $r\omega[j]$ *versus* d, también llamado diagrama P-D [EL94a] (figura 1.2b). Se representan los límites del intervalo de selección U_k y L_K como líneas cuyo origen está en el punto $(0, 0)$. La pendiente de cada una de ellas es $k + \rho$ y $k - \rho$, respectivamente. Las regiones delimitadas por estas líneas son útiles en el análisis de la función de selección

Se define la función de selección (SEL) como

$$q_{j+1} = SEL(\omega[j], d) \tag{1.15}$$

tal que $-d < \omega[j+1] = r \cdot \omega[j] - d \cdot q_{j+1} < d$. La función de selección puede ser representada como un conjunto de subfunciones s_k. $-a \leq k \leq a$ de manera que

$$q_{j+1} = k \quad si \quad s_k \leq r \cdot \omega[j] < s_{k+1} \tag{1.16}$$

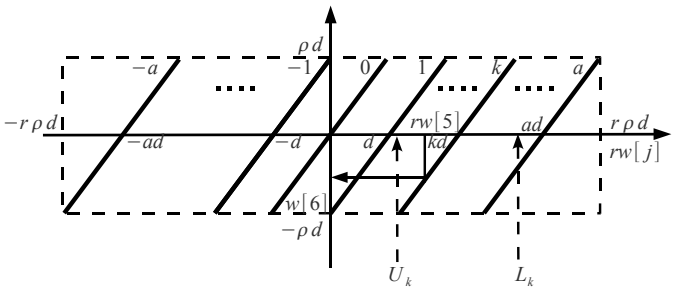
Cada miembro de s_k es una función de d. Las s_k deben de caer en la frontera entre las regiones seleccionadas o, en el caso de conjuntos de dígitos redundantes, en las regiones en las que se solapan.

$$L_k \leq s_k \leq U_{k-1} \tag{1.17}$$

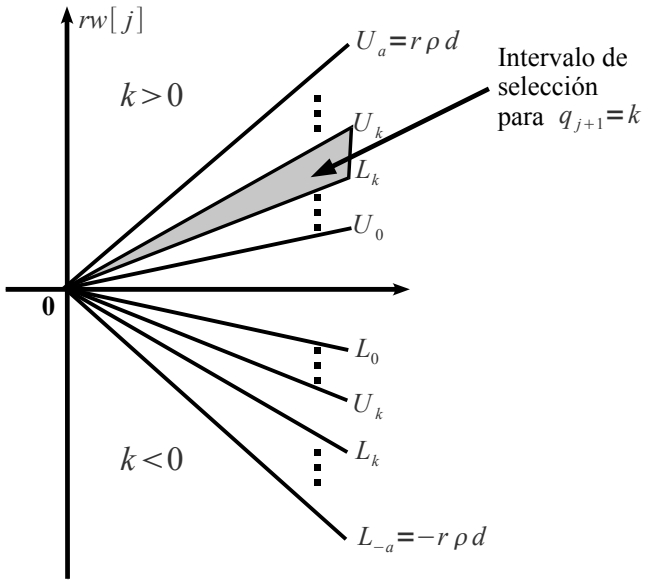
Una de las principales motivaciones para escoger conjuntos de dígitos redundantes es que cuanto mayores sean las regiones de solapamiento, más flexibilidad se tiene para escoger la función de selección.

D. Raíz cuadrada y raíz cuadrada recíproca

El algoritmo DR para raíz cuadrada tiene muchas similitudes con el explicado para división y recíproco. Al igual que en la sección anterior se suponen operandos fraccionales y normalizados. La raíz cuadrada acepta un operando no negativo x y devuelve un resultado no negativo s, donde $|x - s^2| < ulp$ (*ulp: unit*



(a) Diagrama de Robertson



(b) Diagrama P-D

Figura 1.2: Diagramas para el diseño de algoritmos sustractivos

in the last place. El resultado en la iteración j se denota por $s[j]$. La recurrencia en este caso está dada por

$$\omega[j] = r \cdot \omega[j] - 2 \cdot s[j] \cdot s_{j+1} - s_{j+1}^2 \cdot r^{-(j+1)} \tag{1.18}$$

Se define

$$f[j] = 2 \cdot s[j] + s_{j+1} \cdot r^{-(j+1)} \tag{1.19}$$

Entonces,

$$\omega[j+1] = r \cdot \omega[j] - f[j] \cdot s_{j+1} \tag{1.20}$$

Esta recurrencia es muy similar a la obtenida anteriormente para la división. En la práctica $f[j]$ es fácil de generar. Este hecho permite combinar la implementación de la división y raíz cuadrada de una manera sencilla. El formato del resultado es el mismo que para el caso anterior. El factor de redundancia ρ también coincide. Para este caso los límites residuales son:

$$\underline{B} = -2 \cdot \rho \cdot s[j] + \rho^2 \cdot r^{-j} \quad y \quad \bar{B} = 2 \cdot \rho \cdot s[j] + \rho^2 \cdot r^{-j} \tag{1.21}$$

Similarmente, el intervalo de selección para el dígito k sobre el factor de redundancia ρ esta definido por

$$\begin{aligned} U_k &= 2 \cdot s[j] \cdot (k+\rho) + (k+\rho)^2 \cdot r^{-(j+1)} \\ L_k &= 2 \cdot s[j] \cdot (k-\rho) + (k-\rho)^2 \cdot r^{-(j+1)} \end{aligned} \tag{1.22}$$

Estas dos cantidades dependen de j, lo que significa que pueden variar de una iteración a otra. En el caso de la división las cotas y los intervalos de selección son constantes. Esta simplificación no es posible para el caso de la raíz cuadrada.

Es importante destacar que este desarrollo teórico para la explicación de los algoritmos sustractivos se ha hecho en términos de un radix arbitrario r. En la práctica, las implementaciones están limitadas a potencias de 2. Cuanto mayor sea el radix de una operación, mayor es el número de bits generados en cada iteración. Esto reduce el número de iteraciones necesarias para producir un resultado. Sin embargo, el incremento del radix tiene ciertos inconvenientes. La complejidad de la función de selección y de la lógica para generar los factores aumenta mucho, dado que el número de dígitos candidatos a la selección se dobla cada vez que incrementamos el radix. Cuando se intenta acelerar la convergencia lineal de estos algoritmos de este modo la cantidad de hardware necesario se incrementa significativamente. Este es su principal inconveniente.

Esquema	radix 2	radix 4	radix 8	radix 16	radix 512
Tiempo de ciclo	1.0	1.3	1.6	1.6	1.7
Nº ciclos	57	29	20	15	10
Speedup	1.0	1.5	1.8	2.4	3.4
Área	1.0	1.1	1.5	1.7	3.9

Tabla 1.3: Comparación de esquemas con distinto radix respecto de radix 2

E. Implementación

Existen muy diversos ejemplos para ilustrar la implementación de este tipo de algoritmos. Se han realizado diversas implementaciones utilizando radix cada vez mayores. Para radix 2,4 y 8 se pueden realizar implementaciones directas de los algoritmos [EL89, NL99a, NL98]. Si se quiere utilizar un radix mayor la implementación directa del algoritmo no resulta práctica debido a la complejidad de la función de selección. Para estos casos es necesario hacer modificaciones del algoritmo. Se pueden encontrar implementaciones para radix 16 (que consiste en dos etapas de radix 4 solapadas)[NL99b] e incluso para radix muy alto (512) [EL94a].

En general, para este tipo de métodos, el incremento del radix produce un incremento de la velocidad del método por la vía de la reducción del número de ciclos necesarios. Por ejemplo, para el caso de la división es posible establecer una tabla con las distintas figuras de mérito relativas a los requerimientos de una implementación con radix 2 (Tabla 1.3)[EL94a]. Sin embargo, un radix de 2^k produce un incremento más pequeño que el ideal (k) debido a que también se incrementa el tiempo de ciclo. También se produce un incremento significativo del área.

Por ejemplo en [NL99a], se aplica lo explicado en las secciones anteriores para el caso de una unidad de división para doble precisión con radix 4 (Fig. 1.3). Para radix 4 la recurrencia es

$$\omega[j+1] = 4 \cdot \omega[j] - q_{j+1} \cdot d \qquad j = 0, 1, ..., 28 \tag{1.23}$$

con un valor inicial $\omega[0] = x$ y con una función de selección definida del siguiente modo:

$$q_{j+1} = SEL(\hat{y}, d_4) \qquad q_j = \{-2, -1, 0, 1, 2\} \tag{1.24}$$

donde d_4 es d truncado después del cuarto bit fraccional. Solo se necesitan 3 bits para realizar la selección, el MSB es siempre uno debido a la normalización

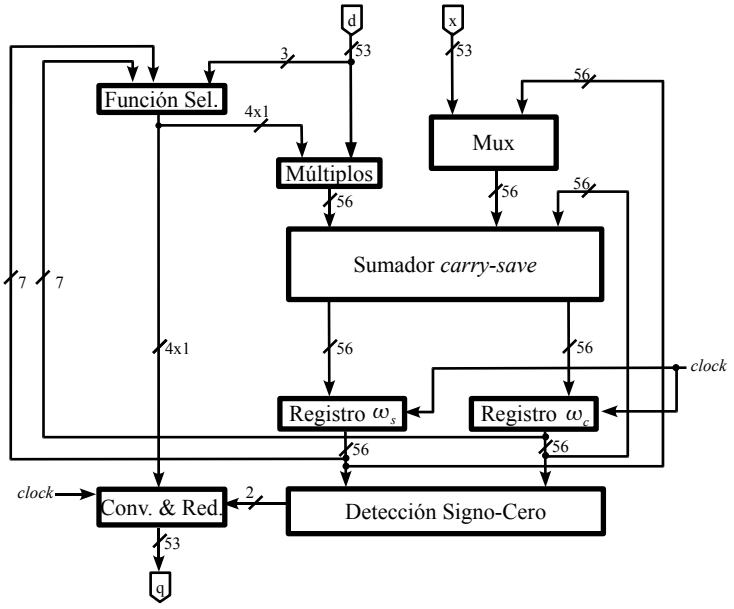


Figura 1.3: Diagrama de bloque para un divisor de radix 4

de d. El resto estimado $\hat{y} = 4 \cdot \omega_{s_4} + 4 \cdot \omega_{c_4}$ se trunca también después del cuarto bit fraccional. De este modo, sumados con los tres bits enteros, la función de selección requiere de 7 bits, tal y como se muestra en la figura **??**. Para que el resto esté siempre acotado, se desplaza x a la derecha en un bit obteniéndose 54 bits para la representación de la mantisa. Se necesita otro bit extra para realizar el redondeo, de modo que se realiza un desplazamiento adicional, quedando 55 bits más un bit de signo para $\omega[j]$.

Cada división requiere 28 ciclos para calcular los dígitos del cociente más un ciclo para inicializar la recurrencia y otro para hacer el redondeo. La figura no incluye un controlador y árbol para distribuir la señal de reloj que también son necesarios. Las cuestiones a abordar para la implementación de unidades para raíz cuadrada son muy similares a las de división. Existen también múltiples ejemplos de dichas implementaciones [LA01, Tak01]. Son muy habituales, también, los diseños en los que se combinan las operaciones de división y raíz

cuadrada [CL94, PZ95]. El diseño conjunto permite un ahorro de recursos hardware.

1.2.2. Algoritmos multiplicativos

Los métodos descritos en esta sección calculan una función mejorando una aproximación inicial de una manera iterativa. Para estas operaciones descritas la operación fundamental es la multiplicación; de hecho reciben el nombre de algoritmos multiplicativos [EL94a]. Estos métodos tienen una convergencia cuadrática, lo que significa básicamente que el número de bits de exactitud se dobla en cada iteración del algoritmo. Como consecuencia de esto, el número de iteraciones para alcanzar la exactitud deseada es menor que en el caso de los algoritmos *digit-recurrence*. Sin embargo, dado que cada iteración involucra una multiplicación, el tiempo dedicado a cada una de ellas es mayor.

Una aplicación especialmente atractiva de estos métodos es su utilización en las unidades de punto flotante de los procesadores. La razón es que pueden ser implementados reusando el multiplicador ya existente de manera que no requieren hardware adicional [FO01]. Este hecho y su rapidez hacen que sean la opción preferida para ser implementada en diseños modernos.

Para realizar implementaciones eficientes es necesario realizar algunas modificaciones en el multiplicador. En general, para división y raíz cuadrada estos algoritmos comienzan con el cálculo de los recíprocos para obtener luego la división o la raíz cuadrada [PB02].

En esta descripción se consideran los operandos fraccionales y normalizados. Por otro lado, el cómputo del signo del signo y los exponentes se hace de forma separada y es muy sencillo para ambos.

A. El algoritmo de Newton-Raphson

El algoritmo de Newton-Raphson (NR) es un método basado en una técnica de resolución de ecuaciones. Consiste en la obtención de un cero de la función, esto es, el valor de ω (que es el resultado que buscamos) para el cual $f(\omega) = 0$ [TF96]. El método se puede ilustrar con la gráfica de la función. Supongamos la figura para una función concreta (Fig. 1.4):

- Se realiza una gráfica aproximada de la función $\delta = f(\omega)$. A esta función se le denomina *función primitiva*.
- Se quiere calcular la raíz donde $f(\omega)$ cruza el eje ω. Se comienza por una primera aproximación (ω_1).

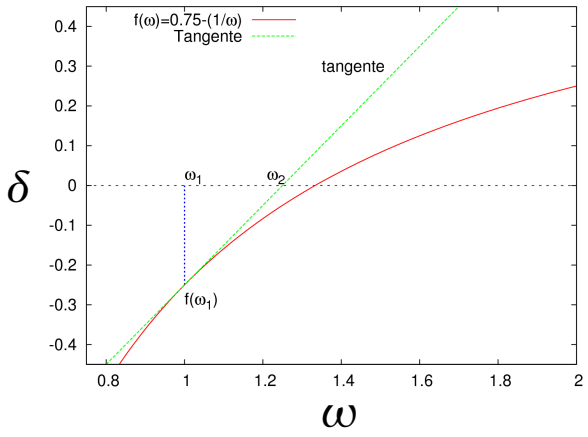


Figura 1.4: Ejemplo: Curva $f(\omega) = 0{,}75 - \dfrac{1}{\omega}$ y su tangente en $f(\omega_1)$

- La siguiente aproximación (ω_2) será el lugar donde la tangente a $f(\omega)$ en el punto $(\omega_1, f(\omega_1))$ cruza el eje ω.
- Según la figura 1.4 la ecuación de esta línea tangente viene dada por:

$$\delta - f(\omega_1) = f'(\omega_1) \cdot (\omega - \omega_1) \tag{1.25}$$

  $f'(\omega_1)$ es la derivada de la función en ese punto.
- La línea tangente cruza el eje ω en $\omega = \omega_2$ y $\delta = 0$.

$$\begin{aligned} 0 - f(\omega_1) &= f'(\omega_1) \cdot (\omega_2 - \omega_1) \\ \omega_2 &= \omega_1 - \frac{f(\omega_1)}{f'(\omega_1)} \end{aligned} \tag{1.26}$$

Así es posible encontrar la siguiente aproximación al resultado en función

de la anterior. De manera más general

$$\omega_{i+1} = \omega_i - \frac{f(\omega_i)}{f'(\omega_i)} \tag{1.27}$$

El método está basado en esta fórmula de recursión

División y Recíproco: La fórmula anterior es una iteración recursiva que puede ser usada para resolver muchas ecuaciones. En este caso el método general se aplica al cálculo del recíproco de X. Para este caso, hay que escoger una función primitiva que tenga un cero en $\omega = 1/X$. La ecuación que se elige es $f(\omega) = \frac{1}{\omega} - X$, que puede ser resuelta usando la recursión (1.27).

Si

$$f(\omega) = \frac{1}{\omega} - X \tag{1.28}$$

entonces, en el punto $\omega = \omega_i$

$$f'(\omega_i) = -(\frac{1}{\omega_i})^2 \tag{1.29}$$

Después de la sustitución, se obtiene la fórmula recursiva específica para el recíproco.

$$\omega_{i+1} = \omega_i \cdot (2 - X \cdot \omega_i) \tag{1.30}$$

En el caso de que la función deseada sea la división (Y/X), no hay más que obtener el recíproco $(1/X)$ tal y como se ha explicado y multiplicarlo por el dividendo (Y).

La recurrencia en la ecuación (1.30) comienza con una aproximación inicial ω_0. Cada iteración requiere dos multiplicaciones y una resta del valor 2. La convergencia de este método es cuadrática.

Supongamos un ejemplo del cálculo del recíproco de $X = 5/8$. Supongamos que empezamos con una aproximación inicial $\omega_0 = 1$. El proceso de obtención de la aproximación consisten en la evaluación de (1.30). El resultado exacto es $1/X = 1{,}6$. En la Tabla 1.4 podemos ver como, en cada iteración, el resultado obtenido se va aproximando al exacto y el error va disminuyendo. Los exponentes de los resultados confirman la convergencia cuadrática del resultado.

j	ω_i	$X \cdot \omega_i$	$2 - X \cdot \omega_i$	ω_{i+1}	*error*
0	1	$5 \cdot 2^{-3}$	$11 \cdot 2^{-3}$	1,375	-0.225
1	$11 \cdot 2^{-3}$	$55 \cdot 2^{-6}$	$73 \cdot 2^{-6}$	1,5683594...	-0.03164...
2	$803 \cdot 2^{-9}$	$4015 \cdot 2^{-12}$	$4177 \cdot 2^{-12}$	1,5993743...	-0.000625706...

Tabla 1.4: Ejemplo del algoritmo de Newton-Raphson para el recíproco

Raíz cuadrada y Raíz cuadrada recíproca: El proceso es muy parecido al caso de la división. Ahora la función primitiva a resolver es

$$f(\omega) = \frac{1}{\omega^2} - X \tag{1.31}$$

Siguiendo el mismo procedimiento establecido por (1.27) se obtiene la recursión para la raíz cuadrada recíproca.

$$\omega i + 1 = \frac{1}{2} \cdot \omega_i \cdot (3 - X \cdot \omega_i^2) \tag{1.32}$$

Implementación: La implementación de estos algoritmos en hardware hace que a veces haya que modificar su representación matemática para hacerla más adecuada. Además es necesario establecer algunas consideraciones en cuanto al rango de los operandos, número de iteraciones, etc.

Es necesario tener en cuenta, tal y como se ha mencionado, que los algoritmos multiplicativos doblan el número de bits correctos de la aproximación al resultado en cada iteración. Este hecho hace que estos algoritmos sean muy rápidos si se emplea un número pequeño de iteraciones. Por eso, una característica común en todas las implementaciones será que todas comienzan usando una aproximación al resultado (*semilla*) para reducir dicho número. Normalmente, la semilla se encuentra almacenada en una tabla. A continuación se realizan las iteraciones del algoritmo (Fig. 1.5).

Así las implementaciones de estos algoritmos se componen de una tabla que contiene las aproximaciones iniciales y el hardware que realiza las iteraciones (ver [EL94a] para más detalles). Se obtendrán diferentes niveles de exactitud dependiendo del número de iteraciones y de la exactitud de la semilla.

División (Y/X) y recíproco ($1/X$): Para la implementación de cualquier algoritmo en hardware es necesario separarlo en operaciones hardware asimilables a una unidad especializada para su cálculo. Como es obvio, la operación principal para este algoritmo será la multiplicación.

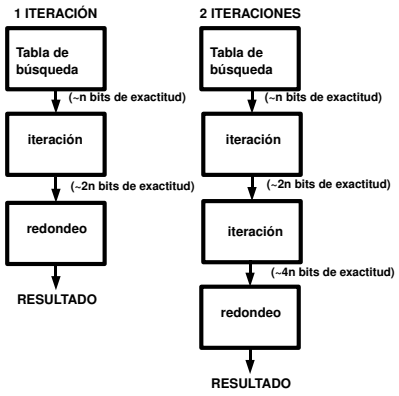


Figura 1.5: Arquitecturas para algoritmos multiplicativos

Basándose en la recursión obtenida en (1.30), las operaciones básicas para una iteración del algoritmo de Newton-Raphson consistirán en las siguientes operaciones aritméticas (Fig 1.6).

$$\begin{aligned} n_{i-1} &= X \cdot \omega_{i-1} \\ d_{i-1} &= 2 - n_{i-1} \\ \omega_i &= \omega_{i-1} \cdot d_{i-1} \end{aligned} \tag{1.33}$$

Esta secuencia de operaciones proporciona una aproximación al recíproco. La primera aproximación ω se obtiene de una tabla. Se necesitan dos multiplicaciones por iteración. Un detalle importante a destacar es que estas dos multiplicaciones son dependientes entre sí. Luego, no pueden ser ejecutadas en paralelo ni tampoco son especialmente adecuadas para su ejecución en un único multiplicador segmentado. Una multiplicación adicional por el dividendo Y proporciona el resultado para la división.

Raíz cuadrada ($\sqrt{X}$) y raíz cuadrada recíproca ($1/\sqrt{X}$): La ecuación de convergencia (1.32) se expande en las siguientes operaciones aritméticas (Fig. 1.6).

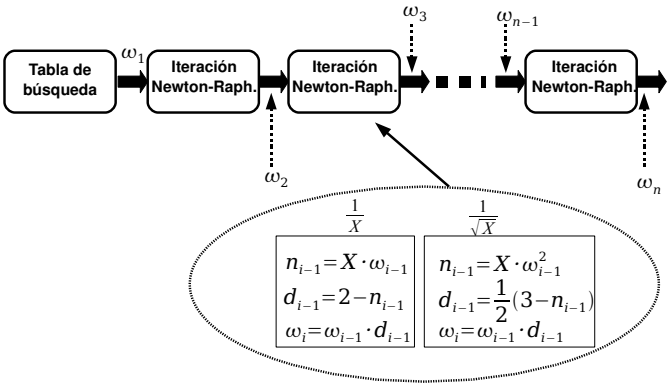


Figura 1.6: Estructura del algoritmo NR

$$\begin{aligned} s_{i-1} &= \omega_{i-1}^2 \\ n_{i-1} &= X \cdot n_{i-1} \\ d_{i-1} &= 3 - n_{i-1} \\ \omega_i &= \frac{\omega_{i-1}}{2} \cdot d_{i-1} \end{aligned} \tag{1.34}$$

En este caso son tres las multiplicaciones dependientes que se realizan por cada iteración. Se necesita una multiplicación adicional por el operando X para obtener el resultado de la raíz cuadrada.

Ejemplo de implementación: Una de las implementaciones más conocidas es una adaptación del algoritmo NR para la operación multiplicación-suma [ITY95]. Este método aborda tres maneras diferentes de obtener la aproximación inicial para el algoritmo. El primero se basa en una aproximación directa (AD) al resultado que está almacenado en una tabla. El segundo consiste en una aproximación lineal (AL) en la que se evalúa una función lineal mediante unos coeficientes obtenidos de unas tablas. La última de ellas es una modificación mejorada de la aproximación lineal (ML), que es la propuesta en el artículo. Para direccionar las tablas se utilizan los m primeros bits del operando X.

La ecuación (1.30) del método se modifica para obtener un algoritmo de un orden de convergencia más alta para una unidad de multiplicación-suma según [Fer67]:

$$\begin{aligned} &paso\ 1: && D_i = 1 - \omega_{i-1} \cdot X \\ &paso\ 2: && \omega_{i+1} = (1 + D_i + D_i^2 + D_i^3 + \cdots + D_i^{K_i - 1}) \cdot \omega_{i-1} \end{aligned} \tag{1.35}$$

Este algoritmo tiene una convergencia de orden k. El algoritmo reescrito se denomina algoritmo HOC (n es el número de iteraciones):

[Algoritmo HOC]
for i=1 to n do
paso 1: $D_{i,0} = 1 - \omega_{i-1} \cdot X$
paso 2: for j=1 to $k_i - 2$ do
$D_{i,j} = D_{i,0} \cdot D_{i,j-1} + D_{i,0}$
paso 3: $\omega_i = \omega_{i-1} \cdot D_{i,k_{i-2}} + \omega_{i-1}$

El algoritmo HOC tiene su mejor rendimiento cuando el grado de convergencia es cúbico. Este trabajo propone también un algoritmo acelerado para acelerar la convergencia (AHOC).

[Algoritmo HOC]
for i=1 to n do
paso 1: $D_{i,0} = 1 - \omega_{i-1} \cdot X$
paso 2: $D_{i,1} = D_{i,0} \cdot (1 + D_{i,0}) + \hat{D}^3_{i,0}$
paso 2.2: for j=2 to $k_i - 2$ do
$D_{i,j} = D_{i,0} \cdot D_{i,j-1} + D_{i,0}$
paso2.3: $\omega_i = \omega_{i-1} \cdot D_{i,k_{i-2}} + \omega_{i-1}$

Todas las operaciones de ambos algoritmos pueden ser realizados con una unidad de multiplicación-suma. Para el caso concreto del algoritmo AHOC, $\hat{D}^3_{i,0}$ es una aproximación a $D^3_{i,0}$ leída de tablas direccionada con los l bits más significativos de $D_{i,0}$. $(1 + D_{i,0})$ puede ser obtenido usando muy poco hardware adicional.

El número de bits correctos obtenidos dependerá del tamaño de las tablas utilizadas para obtener las aproximaciones. Asignando una latencia de tres ciclos para la operación de multiplicación suma se puede obtener los resultados de la Tabla 1.5 para distintas configuraciones de los métodos de aproximación y algoritmos. Por ejemplo, para alcanzar doble precisión (54 bits) con $M = 9$ se necesita $m = 18$ para AD+HOC, $m = 13$ para AL+HOC y $m = 11$ para ML+HOC.

Método	$M=3$	$M=6$	$M=9$	$M=12$	$M=15$	$M=18$
AD+HOC	m	$2m+1$	$3m+2$	$4m+3$	$6m+5$	$9m+8$
AL+HOC	$2m+2$	–	$4m+5$	$6m+8$	$8m+11$	$12m+17$
ML+HOC	$2{,}5m$	–	$5m$	$7{,}5m$	$10m$	$15m$
ML+AHOC	–	–	–	$7{,}5m+l-2$	$10m+l-2$	$15m+2l-4$

Tabla 1.5: Número de bits correctos usando m bits para aproximación inicial con M ciclos

B. El algoritmo de Goldschmidt

En este apartado se describe el algoritmo de Goldschmidt (GLD) para división, recíproco, raíz cuadrada y raíz cuadrada recíproca [Gol64]. Se deriva de la expansión en serie de Taylor [Sco85]. Como todos los algoritmos multiplicativos tiene una convergencia cuadrática, tal y como se verá más adelante.

División y recíproco: Para obtener el algoritmo supongamos que el divisor (X) y el dividendo (1) son el numerador y el denominador de una fracción. Esta técnica consiste en dos recurrencias multiplicativas, una de las cuales converge hacia uno, de manera que la otra debe converger hacia la función deseada.

$$\frac{1}{X} \times \frac{K_1}{K_1} \times \frac{K_2}{K_2} \times \cdots \times \frac{K_n}{K_n} \tag{1.36}$$

De manera que

$$K_1 \cdot K_2 \cdots K_n \Rightarrow Cociente \tag{1.37}$$

y

$$X \cdot K_1 \cdot K_2 \cdots K_n \Rightarrow 1 \tag{1.38}$$

La selección del factor K_i es la parte esencial del procedimiento y está basada en lo siguiente. Aunque se pueden utilizar otros rangos, para facilitar esta explicación vamos a suponer que el divisor se puede expresar como

$$X = 1 - \alpha \tag{1.39}$$

donde $\alpha \leq 1/2$ de manera que X será una fracción en punto flotante de la forma $0{,}1xxxxx\cdots$.

Si además se considera K_1 como $1+\alpha$, podemos definir la siguiente cantidad.

$$r_1 = X \cdot K_1 = (1 - \alpha) \cdot (1 + \alpha) = 1 - \alpha^2 \tag{1.40}$$

Donde $\alpha^2 \leq 1/4$ dado que $\alpha \leq 1/2$. Debido a esto este nuevo denominador es de la forma $0{,}11xxxx\cdots$.

Del mismo modo, seleccionando $K_2 = 1 + \alpha^2$ se doblará el número de bits igual a 1 en la parte más significativa de r_2.

$$r_2 = r_1 \cdot K_2 = (1 - \alpha^2) \cdot (1 + \alpha^2) = (1 - \alpha^4) = 0{,}1111xxxx\cdots \tag{1.41}$$

En general, para una iteración i, $\alpha_i < 1/2^n$ y $\alpha_{i+1} < 1/2^2 n$. Continuando esta serie de multiplicaciones hasta que α_i es menor que el bit menos significativo del denominador, obtendremos un denominador en (1.32) equivalente a 1 ($0{,}11111\cdots111$).

Es también importante observar que el multiplicador para cada iteración se puede obtener mediante el complemento a 2 del denominador.

$$K_{i+1} = 2 - r_i = 2 - (1 - \alpha_n) = 1 + \alpha_n \tag{1.42}$$

Paralelamente, se puede obtener el cociente (ω) mediante el cálculo del numerador en la expresión (1.32)

$$\omega_i = K_1 \cdot K_2 \cdots K_i \tag{1.43}$$

Este cociente, tal y como se ha planteado hasta ahora produce como resultado el recíproco del operando X. El proceso para obtener la división es totalmente análogo. Se plantea la misma serie de multiplicaciones pero con un dividendo Y.

$$\frac{Y}{X} \times \frac{K_1}{K_1} \times \frac{K_2}{K_2} \times \cdots \times \frac{K_n}{K_n} \tag{1.44}$$

La obtención de los factores es igual a la del recíproco en la ecuación (1.22). La obtención del resultado viene dado por la siguiente ecuación.

$$\omega_i = Y \cdot K_1 \cdot K_2 \cdots K_i \tag{1.45}$$

Supongamos un ejemplo del cálculo del recíproco de $X = 5/8$ y que la aproximación inicial al resultado es $\omega_0 = 1$. El proceso de obtención consiste en la evaluación de las ecuaciones (1.43), (1.42) y (1.41). El resultado exacto es $X = 1,6$. En la tabla 1.6 se puede ver como, en cada iteración, el resultado se va aproximando rápidamente al resultado exacto y el error decrece.

j	ω_i	r_i	K_{i+1}	ω_{i+1}	*error*
0	1	$5 \cdot 2^{-3}$	$11 \cdot 2^{-3}$	1,375	-0.225
1	$11 \cdot 2^{-3}$	$55 \cdot 2^{-6}$	$73 \cdot 2^{-6}$	1,5683594...	-0.03164...
2	$803 \cdot 2^{-9}$	$4015 \cdot 2^{-12}$	$4177 \cdot 2^{-12}$	1,5993743...	-0.000625706...

Tabla 1.6: Ejemplo del algoritmo de Goldschmidt para el recíproco

Raíz cuadrada y raíz cuadrada recíproca: En este caso se utiliza el algoritmo GLD para obtener una aproximación de la raíz cuadrada ($\sqrt{X}$) o la raíz cuadrada recíproca ($1/\sqrt{X}$) del operando X. Al igual que en el caso anterior, consiste en encontrar la serie de factores multiplicativos en la siguiente fracción

$$\frac{1}{X} \times \frac{K_1}{K_1^2} \times \frac{K_2}{K_2^2} \times \cdots \times \frac{K_n}{K_n^2} \tag{1.46}$$

El denominador al igual que en el caso anterior tiende a 1.

$$X \cdot K_1^2 \cdot K_2^2 \cdots K_n^2 \Rightarrow 1 \tag{1.47}$$

El producto $\omega_n = K_1 \cdot K_2 \cdots K_n$ proporciona una aproximación a $1/\sqrt{X}$. Para obtener una aproximación a la raíz cuadrada de X será necesario calcular el producto $K_1 \cdot K_2 \cdots K_n$.

También es muy importante en este caso la selección de los factores subsiguientes K_i, que en este caso es un poco más complicada que en el caso anterior.

Suponiendo un operando de la forma

$$X = 1 - \alpha \tag{1.48}$$

y la selección de un $K_1 = 1 + \alpha$, obtenemos

$$r_1 = X \cdot K_1^2 = (1 - \alpha) \cdot (1 + \alpha)^2 \tag{1.49}$$

En este caso, el valor del siguiente factor se plantea como una aproximación

$$K_{i+1} \approx \frac{3 - r_i}{2} \tag{1.50}$$

Implementación: El último aspecto a tratar es la implementación del algoritmo GLD. Las consideraciones generales, en cuanto al rango de los operandos y de la aproximación inicial, son las mismas que en el caso del algoritmo de Newton-Raphson.

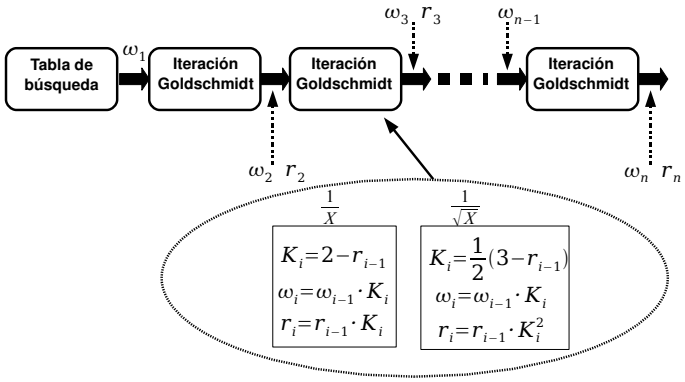


Figura 1.7: Estructura del algoritmo GLD

División (Y/X) y recíproco ($1/X$): Como se explicó anteriormente, el algoritmo consiste básicamente en encontrar una secuencia de números $K_1, K_2, \cdots K_i$ que satisfagan la relación

$$r_i = X \cdot K_1 \cdot K_2 \cdots K_i \rightarrow 1 \tag{1.51}$$

El primer elemento K_1 es la aproximación de baja precisión al recíproco (semilla). K_i se usa en el cómputo de r_i. Después de esto se calcula un nuevo K_{i+1}.

$$K_i = 2 - r_{i-1} \tag{1.52}$$

El resultado después de i iteraciones para el recíproco es:

$$\omega_i = K_1 \cdot K_2 \cdots K_i \rightarrow 1/X \tag{1.53}$$

Las operaciones aritméticas básicas para cada iteración se representan en la figura 1.7. Como se puede comprobar en este caso son igualmente necesarias dos multiplicaciones por iteración. Pero con una diferencia. Ahora estas dos multiplicaciones son independientes de manera que pueden ser realizadas en paralelo o en un mismo multiplicador segmentado.

En cada iteración del algoritmo se obtiene la nueva aproximación al resultado mediante el cálculo de un nuevo K. Como se muestra en la figura 1.7, cada iteración consisten en las siguientes operaciones:

$$\begin{aligned} K_i &= 2 - r_{i-1} \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i \end{aligned} \tag{1.54}$$

El algoritmo NR sólo es capaz de obtener el recíproco, de modo que para calcular la división se calcula una estimación para el recíproco y luego se multiplica por el dividendo Y. Sin embargo, para el algoritmo GLD hay dos maneras distintas de llegar al resultado para la división. La primera de ellas es igual a la del algoritmo NR. La otra posibilidad es partir de una aproximación inicial a la división. Haciendo esto el algoritmo converge directamente hacia la división.

Raíz cuadrada ($\sqrt{X}$) y raíz cuadrada recíproca ($1/\sqrt{X}$): Nuevamente, se trata de encontrar una secuencia $K_1, K_2, \cdots K_i$ que ahora satisface la relación

$$r_i = X \cdot K_1^2 \cdot K_2^2 \cdots K_i^2 \rightarrow 1 \tag{1.55}$$

En cada nueva iteración se calcula otro K.

$$K_i = \frac{3}{2} - \frac{r_{i-1}}{2} \tag{1.56}$$

Se obtiene una aproximación a la raíz cuadrada recíproca por medio de la expresión

$$\omega_i = K_1 \cdot K_2 \cdots K_i \rightarrow 1/\sqrt{X} \tag{1.57}$$

Las multiplicaciones necesarias en cada iteración serán ahora tres (Fig. 1.7) y dos de ellas independientes entre sí. Vuelven a existir dos posibilidades para el cálculo de la raíz cuadrada. Mediante una multiplicación por el operando X o partiendo, desde el principio, de una aproximación a la raíz cuadrada.

Cada iteración del algoritmo para esta operación se compone de las siguientes operaciones:

$$\begin{aligned} K_i &= \frac{1}{2}(3 - r_{i-1}) \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i^2 \end{aligned} \tag{1.58}$$

1.3. Redondeo de algoritmos multiplicativos

El redondeo de los algoritmos DR es muy sencillo puesto que en cada iteración se calcula el resto parcial del resultado. En estos algoritmos el resultado se calcula dígito a dígito en cada iteración. Tradicionalmente el redondeo de estos algoritmos se realiza tal y como fue descrito en [Fan87] y [EL92] en base a los siguientes pasos:

- Cálculo del resultado con un dígito adicional para poder realizar el redondeo
- Se necesita primero una corrección del resultado cuando el resto es negativo para producir un resultado de resto positivo. Esto se hace mediante la detección del signo del resto y la suma de un bit si es necesario.
- Redondeo del resultado teniendo en cuenta si el resto está normalizado o no.

Sin embargo, para los algoritmos multiplicativos, es necesario calcular el resto para realizar el redondeo puesto que no lo producen de manera directa. Existe un método que es una excepción a este criterio [Mar90, IM99]. Dicho método evita el cálculo del resto, pero es menos eficiente puesto que se necesita calcular el resultado con el doble de exactitud que la requerida finalmente. Se ha demostrado que se puede obtener el resultado correctamente redondeado si se calcula el resultado con el doble de exactitud [Mar90]. Esto supone el cálculo de una iteración adicional del algoritmo y una mayor ancho del multiplicador utilizado. De esta manera, el consumo en hardware y tiempo del algoritmo es sensiblemente mayor.

La estrategia tradicional usada para redondear el resultado de estos algoritmos es crear un resultado preliminar que es una aproximación al resultado. Denotamos el resultado exacto (con precisión infinita) como ω. Después del redondeo, se debe obtener una aproximación al resultado de n bits que esté *correctamente redondeada*. Esto quiere decir que el resultado obtenido es el mismo que el que se hubiese obtenido de redondear el resultado exacto ω. La aproximación obtenida directamente del algoritmo se denotará como $\hat{\omega}$ que tendrá algunos bits extra de exactitud. Se supone, además, que $\hat{\omega}$ tendrá un error bipolar (*two-sided*), es decir, un error que puede ser positivo o negativo, como es usual en las implementaciones hardware.

El método descrito a continuación obtiene una nueva aproximación ω' modificando $\hat{\omega}$. Esta nueva aproximación tiene n más un número determinado de bits

que depende del caso. Estos bits adicionales y el signo del resto son usados para determinar la acción de redondeo que se aplica a ω' para obtener el resultado correctamente redondeado. Este resultado se usa para calcular el resto de la operación. El signo del resto y los últimos bits de la aproximación se usan para decidir qué acción de redondeo realizar.

$$\omega' \approx \frac{Y}{X}$$

$$rem = Y - \omega' \cdot X$$

$$compara\ (\omega' \cdot X)\ con\ Y$$

El cálculo del resto y su comparación con cero se sustituye con el cálculo de $(\hat{\omega} \cdot X)$ y su comparación con el dividendo. Es importante resaltar aquí que sólo unos pocos bits de $(\hat{\omega} \cdot X)$ son significativos para esta operación

Por simplicidad, se expone aquí únicamente el redondeo al más próximo, el redondeo para el resto de los modos puede ser revisado en [Sch95, OF97]. Las conclusiones obtenidas para redondeo al más próximo pueden ser aplicadas a los otros tres modos de redondeo.

En esta sección se presentan el método más conocido [Sch95] y una extensión de él [OF97] para el redondeo de algoritmos multiplicativos. Ambos intentan evitar el cálculo del resto en el mayor número de casos posibles.

1.3.1. Método clásico de redondeo

Supongamos que el resultado $\hat{\omega}$ se calcula con $n + k$ bits de precisión y que tiene una exactitud de $\pm 2^{-(n+j)}$ donde $k \geq j \geq 2$.

$$-2^{-(n+j)} < \omega - \hat{\omega} < 2^{-(n+j)} \tag{1.59}$$

Este método está ilustrado en la figura 1.8 para el caso concreto con $j = 2$ y $k = 5$. En la parte superior de la figura se representan los posibles valores que pueden tomar $\hat{\omega}$ si se calcula con $n + 5$ bits de precisión. No todos los puntos están etiquetados. Los que sí lo están están representados por los bits $\hat{\omega}[n : n+2]$ (los bits $\hat{\omega}[n+3 : n+5]$ son cero para estos puntos). L es el bit menos significativo a la precisión requerida, $\hat{\omega}[n]$. Los otros dos bits representados son los bits extra obtenidos para calcular $\hat{\omega}$ a la exactitud requerida. En la parte inferior se representan los posibles valores para ω' que tiene un bit extra de precisión y también de exactitud.

El método consiste en cuatro pasos:

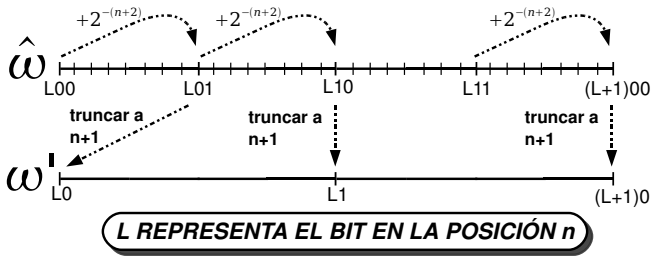


Figura 1.8: Dos primeros pasos del redondeo

$\omega'[n+1]$	resto	acción de redondeo
0	–	truncar
1	=0	–
1	-	**truncar**
1	+	**incrementar**

Tabla 1.7: Tabla de redondeo para redondeo al más próximo

- Incrementar $\hat{\omega}$ en una constante menor que $\pm 2^{-(n+1)}$ pero mayor o igual que $\pm 2^{-(n+j)}$. En este caso particular $\pm 2^{-(n+2)}$.
- Para obtener ω' truncar la sobreestimación obtenida a $n+1$ bits.
- Determinar si el cálculo del resto es necesario. Esto se hace examinando el bit $\omega'[n+1]$ en la Tabla 1.7.
- Calcular el resto $rem = 1 - X \cdot \omega'$ (si es necesario). Observando el signo el signo del resto (si es necesario) y el bit $n+1$ de ω' (Tabla 1.7) realizar la acción de redondeo apropiada.

Este método [Sch95] calcula una nueva aproximación ω' con $n+1$ bits de precisión y una exactitud de $\pm 2^{-(n+1)}$ mediante la transformación de $\hat{\omega}$. La acción de redondeo se determina examinando el bit $n+1$ de ω' y el signo del resto. El cálculo del resto no es necesario en todos los casos. Las diferentes acciones de redondeo están recogidas en la tabla de redondeo (Tabla 1.7). Cuando el bit $n+1$

de ω' es cero la acción no depende del valor del resto. Entonces, para estos casos no es necesario calcularlo. Usando este método la mitad de las entradas posibles de la tabla para $\omega'[n+1]$ necesitan el cálculo del resto. Como se verá más adelante, se determinará que esto supone un 25 % de los casos cuando ω' se calcula con un bit extra.

1.3.2. Generalización del método clásico

Existe una generalización al método anterior [OF97] que consiste en obtener ω' con m bits extra ($m > 1$). El método anteriormente descrito requiere obtener la aproximación al resultado con un bit de guarda adicional, lo que permite evitar el cálculo del resto para la mitad de las entradas posibles de la tabla. Este método puede ser extendido del siguiente modo. Considérese por ejemplo, que la aproximación $\hat{\omega}$ se obtiene con $n+3$ bits de exactitud.

$$-2^{-(n+3)} < \omega - \hat{\omega} < 2^{-(n+3)} \tag{1.60}$$

Los dos primeros pasos son muy similares al caso anterior:

- Incrementar $\hat{\omega}$ en $2^{-(n+3)}$.
- Truncar el resultado obtenido a $n+2$ bits para formar ω'. Ahora tiene $n+2$ bits de precisión y de exactitud.
- Determinar si el cálculo del resto es necesario según la tabla 1.8
- Cálculo del resto, si es necesario, y determinación de la acción de redondeo (Tabla 1.8)

Las acciones de redondeo para este caso se muestra en la Tabla 1.8. En este caso sólo una entre cuatro entradas posibles de la tabla necesitan el cálculo del resto. Estas combinaciones están marcadas en negrita.

Este método puede ser generalizado para cualquier número de bits de guarda adicionales m, con $m \geq 1$:

- Incrementar $\hat{\omega}$ en $2^{-(n+m+1)}$.
- Truncar la sobreestimación a $n+m$ bits para obtener ω'. El error será ahora $\pm 2^{-m} \cdot ulp$.
- Determinar si el cálculo del resto es necesario.

$\omega'[n+1]$	resto	acción de redondeo
00	=0	truncar
00	-	truncar
00	+	truncar
01	=0	truncar
01	-	truncar
01	+	truncar
10	=0	–
10	-	**truncar**
10	+	**incrementar**
11	=0	incrementar
11	-	incrementar
11	+	incrementar

Tabla 1.8: Tabla de redondeo con dos bits de guarda

- Cálculo del resto y determinación de la acción de redondeo.

En este caso, cuando usamos m bits de guarda adicionales, el cómputo del resto será necesario para 2^{-m} de todas los posibles combinaciones de estos bits. Tal y como se verá más adelante (capítulo 3) cuando $m = 1$ se realizará el redondeo en el 25 % de los casos mientras que, cuando $m = 2$ se realiza en el 12,5 % de ellos. Para la implementación del redondeo en los casos en los que se necesita el resto es necesaria una multiplicación adicional para calcularlo. Esto puede suponer en algunos casos hasta un 25 % de la latencia total del algoritmo

Capítulo 2

Optimización del tamaño del multiplicador

Todas las operaciones aritméticas están afectadas por un error cuando son implementadas en hardware. Este error es debido en parte a que, en hardware, se dispone de una precisión finita para la representación de los números. Los algoritmos multiplicativos, dado que son un conjunto de operaciones aritméticas, están también afectados por este tipo de errores. Este capítulo presenta un análisis exhaustivo de las distintas contribuciones existentes al error total en su implementación hardware. Dicho análisis se realiza para las cuatro operaciones: recíproco, división, raíz cuadrada inversa y raíz cuadrada. El resultado de este análisis son expresiones analíticas que permiten obtener el error del resultado de estas operaciones. Dicho error se obtiene en función del error de la iteración anterior y el error debido al cálculo de las operaciones intermedias en cada iteración. La exactitud de estas expresiones es validado por medio de simulaciones exhaustivas. Las mismas expresiones son usadas para, relacionándolas con el tamaño del multiplicador, obtener su tamaño óptimo. A través de este método conseguimos un método de diseño de unidades aritméticas de algoritmos multiplicativos. Encontrar un tamaño óptimo del multiplicador supone un ahorro de tiempo de cálculo y área.

2.1. Introducción

A pesar de que ya existe algún análisis del error para algún algoritmo multiplicativo concreto (ver por ejemplo [ES03a]), no existen análisis que permitan el cálculo del error de una manera precisa y general para todos los algoritmos multiplicativos. Los algoritmos multiplicativos, tal y como indica su nombre, tienen como operación fundamental la multiplicación. La precisión finita disponible en las implementaciones hardware hace que, en cada operación aritmética, se introduzca un error adicional en el cómputo del resultado final. Este error será denominado a partir de aquí como error de las operaciones intermedias o error del hardware. En el caso de los algoritmos multiplicativos, se parte de una aproximación al resultado con una exactitud determinada. Las iteraciones del algoritmo permiten obtener una aproximación final más exacta pero con un determinado error, que se acumula al anterior. A este error se denominará error de aproximación del algoritmo

Partiendo del error de la aproximación inicial y acumulando los errores debido a las distintas operaciones intermedias en la primera iteración, se llega al error total del resultado en la primera iteración. Una vez obtenido éste, se generalizará este resultado para una iteración genérica i, de manera que se obtiene una expresión general para cualquier iteración del algoritmo. El error del resultado de cada iteración es la suma del error de aproximación más el error debido a las operaciones intermedias.

Como se mencionó anteriormente, el error de aproximación se obtiene en función del error del resultado de la iteración anterior y de los errores de las operaciones intermedias. El error de aproximación puede ser expresado en función del número de bits de la aproximación inicial que denotaremos con b. Del mismo modo, el error de las operaciones intermedias tiene relación directa con el tamaño de palabra de las operaciones intermedias (p). Por tanto, lo que se obtiene es la expresión del error total del resultado de cada iteración en función del número de bits de la aproximación inicial y el tamaño del multiplicador.

El número de bits de la aproximación inicial es un dato conocido, y el error del resultado final es un requerimiento del diseño. Es necesario obtener el resultado con una exactitud suficiente para poder realizar el redondeo. Además, es posible establecer un valor máximo y un valor mínimo para el error de cada una de las operaciones intermedias. Utilizando esta información y aplicándola a la expresión del error de una iteración determinada es posible establecer un valor máximo y mínimo para el error de dicha iteración. Estos valores estarán expresados en función del tamaño de palabra de las operaciones intermedias.

Con todos estos elementos se consigue un método de diseño para algoritmos

multiplicativos. Fijando el tamaño de la aproximación inicial y con el requerimiento de exactitud para el resultado, dependiendo de la precisión, se obtiene el tamaño de palabra óptimo de las operaciones intermedias. De este modo, se fija el tamaño óptimo del multiplicador. Esta son la tres magnitudes principales que definen una unidad aritmética basada en este tipo de algoritmos.

2.2. Análisis del error del algoritmo GLD

Antes de realizar el análisis, es necesario fijar las convenciones de notación acerca del error. Se define el error absoluto ε_A como la diferencia entre la magnitud calculada $\hat{A}$ y su valor exacto A.

$$\varepsilon_A = \hat{A} - A \tag{2.1}$$

Entonces,

$$\begin{aligned} \varepsilon_A > 0 &\Rightarrow \hat{A} > A \\ \varepsilon_A < 0 &\Rightarrow \hat{A} < A \end{aligned} \tag{2.2}$$

El resultado de una iteración general i de un algoritmo se denota como $\hat{\omega}_i$. De acuerdo con 2.1, $\hat{\omega}_i$ es la suma del resultado exacto ω más el error correspondiente ε_i [1]. El error está causado por el error de aproximación del propio algoritmo y el error de redondeo de las operaciones intermedias, debido a la precisión finita de la representación de los números en hardware.

Durante todo este análisis no se considerará ningún error en los operandos X e Y, que estarán normalizados y con una precisión de n bits fraccionales. El análisis descrito en este capítulo será el del algoritmo GLD. El análisis para el algoritmo NR es análogo y se incluye en el Apéndice A. El proceso consiste en reescribir las ecuaciones del algoritmo (en este caso las ecuaciones definidas para la implementación del algoritmo GLD en la sección 1.2.2) pero teniendo en cuenta los errores de la operaciones intermedias.

Se asumirá que el error en la aproximación inicial corresponderá a una unidad en el último lugar (1 *ulp*) con respecto a su precisión, es decir, $\pm 2^{-b}$. En el caso de las operaciones intermedias el error correspondiente será 1 *ulp* con respecto a la precisión con que se obtiene su resultado ($\pm 2^{-p}$).

[1] El error contiene las contribuciones del error de aproximación del algoritmo y de las operaciones intermedias. Por eso aparece ω en lugar de ω_i

2.2.1. Recíproco

El punto de partida de este análisis es el error de la aproximación inicial. Como se ha mencionado anteriormente, ésta es normalmente obtenida de la lectura de tablas. La lectura de la aproximación inicial se denominará iteración 0. Supongamos que tiene una precisión y una exactitud de b bits. Luego, el error del resultado de esta primera iteración es

$$\varepsilon_0 = 2^{-b} \tag{2.3}$$

Para hacer el análisis de la siguiente iteración se usan las ecuaciones para el algoritmo GLD de la sección 1.2.2.

$$\begin{aligned} K_i &= 2 - r_{i-1} \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i \end{aligned} \tag{2.4}$$

Estas ecuaciones son ahora reescritas teniendo en cuenta las distintas contribuciones al error de cada operación intermedia. La primera de las ecuaciones se obtiene de la combinación de las ecuaciones (1.44) y (1.46). Entonces, para la iteración 0 obtendremos.

$$\hat{r}_0 = X \cdot \hat{\omega}_0 + \varepsilon^t_{r_0} \tag{2.5}$$

Esta expresión incluye los errores de r_0 y ω_0 usando la notación establecida en (2.1) y el error producido por la multiplicación, $\varepsilon^t_{r_0}$. De ahora en adelante, el superíndice t significará que el error al que se refiere es debido a una operación aritmética intermedia. El subíndice representará la magnitud calculada con esta operación. Es decir, por ejemplo, $\varepsilon^t_{r_0}$ es el error debido a la multiplicación necesaria para calcular r_0. La expresión anterior puede ser expandida teniendo en cuenta el error ω_0 (ε_0).

$$\hat{r}_0 = X \cdot (\omega + \varepsilon_0) + \varepsilon^t_{r_0} \tag{2.6}$$

El siguiente paso es la obtención de K a partir de la ecuación (2.6) para la siguiente iteración del algoritmo, ($\hat{K}_1$). Esto se hace reescribiendo la ecuación (1.45). La operación necesaria en este caso es el complemento a dos de r_0. Se considera que dicha operación también introduce un error.

$$\hat{K}_1 = 2 - \hat{r}_0 + \varepsilon^t_{K_1} \tag{2.7}$$

Al igual que la anterior ecuación, esta puede ser expandida introduciendo el valor de la ecuación (2.6). Se obtiene así el valor de $\hat{K}_1$ calculado con todas las contribuciones al error.

$$\hat{K}_1 = 2 - X \cdot (\omega + \varepsilon_0) - \varepsilon^t_{r_0} + \varepsilon^t_{K_1} \tag{2.8}$$

Una vez que se conoce la expresión completa de $\hat{K}_1$, es posible el cálculo de r para la siguiente iteración ($\hat{r}_1$), que será usado en la siguiente iteración.

$$\hat{r}_1 = \hat{r}_0 \cdot \hat{K}_1 + \varepsilon^t_{r_1} \tag{2.9}$$

Ahora también es posible obtener la aproximación al resultado para la siguiente iteración. Esta expresión es la utilizada para la obtención del error del resultado:

$$\hat{\omega}_1 = \hat{\omega}_0 \cdot \hat{K}_1 + \varepsilon^t_{\omega_1} \tag{2.10}$$

Expandiendo $\hat{\omega}_0$ e introduciendo la expresión obtenida en (2.8) en la ecuación anterior se obtiene la aproximación al resultado de la iteración 1 con todas sus contribuciones al error.

$$\hat{\omega}_1 = (\omega + \varepsilon_0) \cdot [2 - X \cdot (\omega + \varepsilon_0) - \varepsilon^t_{r_0} + \varepsilon^t_{K_1}] + \varepsilon^t_{\omega_1} \tag{2.11}$$

Teniendo en cuenta que $\hat{\omega_1} = \omega + \varepsilon_1$ en (2.11), el error del resultado del algoritmo para la iteración 1 (ε_1) es:

$$\varepsilon_1 = -\frac{1}{\omega} \cdot \varepsilon_0^2 - \omega \cdot (\varepsilon^t_{r_0} - \varepsilon^t_{K_1}) - \varepsilon_0 \cdot (\varepsilon^t_{r_0} - \varepsilon^t_{K_1}) + \varepsilon^t_{\omega_1} \tag{2.12}$$

Siguiendo los mismos pasos, es posible obtener la expresión del error del resultado para las iteraciones siguientes. Podemos establecer, de modo similar, la expresión del error del resultado para una iteración genérica i. Realizando, de nuevo, la combinación de las ecuaciones (1.44) y (1.46) para un iteración $i-1$, se obtiene la siguiente ecuación:

$$\hat{r}_{i-1} = X \cdot \hat{\omega}_{i-1} + \varepsilon^t_{r_{i-1}} \tag{2.13}$$

Igual que en el caso concreto anterior, esta expresión puede ser expandida teniendo en cuenta el error del resultado en la iteración anterior.

$$\hat{r}_{i-1} = X \cdot (\omega + \varepsilon_{i-1}) + \varepsilon^t_{r_{i-1}} \tag{2.14}$$

Volviendo a reescribir la ecuación (1.45) se calcula K para la próxima iteración del algoritmo ($\hat{K}_i$).

$$\hat{K}_i = 2 - X \cdot (\omega + \varepsilon_{i-1}) - \varepsilon^t_{r_{i-1}} + \varepsilon^t_{K_i} \tag{2.15}$$

Es posible ahora el cálculo de $\hat{r}_i$ para la iteración i-ésima.

$$\hat{r}_i = \hat{r}_{i-1} \cdot \hat{K}_i + \varepsilon^t_{r_i} \tag{2.16}$$

La expresión que nos sirve para el cálculo del error de la aproximación al resultado es:

$$\hat{\omega}_i = \hat{\omega}_{i-1} \cdot \hat{K}_i + \varepsilon^t_{\omega_{i-1}} \tag{2.17}$$

Reemplazando (2.15) en (2.17) se obtiene el resultado para la aproximación i-ésima.

$$\hat{\omega}_i = (\omega + \varepsilon_{i-1}) \cdot [2 - X \cdot (\omega + \varepsilon_{i-1}) - \varepsilon^t_{r_{i-1}} + \varepsilon^t_{K_i}] + \varepsilon^t_{\omega_i} \tag{2.18}$$

Teniendo en cuenta que $\hat{\omega}_i = \omega + \varepsilon_i$ en la ecuación anterior, se obtiene el error del resultado para la iteración i-ésima, ε_i.

$$\varepsilon_i = -\frac{1}{\omega} \cdot \varepsilon^2_{i-1} - \omega \cdot (\varepsilon^t_{r_{i-1}} - \varepsilon^t_{K_i}) - \varepsilon_{i-1} \cdot (\varepsilon^t_{r_{i-1}} - \varepsilon^t_{K_i}) + \varepsilon^t_{\omega_i} \tag{2.19}$$

El error del resultado en cada iteración depende de los errores debido a las multiplicaciones en la iteración ($\varepsilon^t_{\omega_i}$,$\varepsilon^t_{r_{i-1}}$), el error debido al complemento a dos ($\varepsilon^t_{K_i}$) y el error del resultado en la iteración previa (ε_{i-1}).

La expresión del error para la división es muy similar a la ecuación (2.19) porque sólo se necesita una multiplicación adicional por el dividendo Y. Por esta razón y por simplicidad a partir de ahora se presentarán únicamente los análisis para el recíproco.

Un análisis más detallado de la expresión del error permite extraer información adicional. El primer término ($-\frac{1}{\omega}\varepsilon^2_{i-1}$) es la expresión bien conocida del error de aproximación del algoritmo de GLD [PB02], que es siempre negativa[2]. El resto de los términos involucra al error de las operaciones intermedias. Estos términos pueden hacer el error total positivo en algunos casos. De hecho, a medida que se incrementa la precisión utilizada, estos términos se hacen cada vez

[2]Hay que tener en cuenta que no se tiene en cuenta el signo de los operandos, siempre se consideran positivos. Los signos se computan de manera separada

más pequeños y el error tiende a tener únicamente la contribución del error de aproximación; igual que en el caso de precisión infinita.

2.2.2. Raíz cuadrada recíproca

Se puede hacer un análisis similar para la raíz cuadrada recíproca y la raíz cuadrada. La aproximación inicial también se lee de tablas. A esta acción se le denotará como la iteración 0. Se supone que la aproximación inicial ($\hat{\omega}_0$) tiene b bits correctos. Entonces, el error de la aproximación inicial es

$$\varepsilon_0 = 2^{-b} \tag{2.20}$$

Igual que para la división y el recíproco, para obtener el error del resultado de una iteración determinada se reescriben, teniendo en cuenta el error de las operaciones intermedias, las ecuaciones de la iteración:

$$\begin{aligned} K_i &= \frac{1}{2}(3 - r_{i-1}) \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i^2 \end{aligned} \tag{2.21}$$

Para obtener el error del resultado en la iteración 1, se comienza con la combinación de las ecuaciones (1.47) y (1.49) para calcular $\hat{r_0}$.

$$\hat{r}_0 = X \cdot (\hat{\omega}^2 + \varepsilon^t_{\omega_0^2}) + \varepsilon^t_{r_0} \tag{2.22}$$

Al igual que para el recíproco no se considera ningún error en los operandos. Ahora, aparece un término adicional en (2.22) respecto del recíproco porque hay una multiplicación adicional para calcular $\hat{\omega}_0^2$ en $\hat{r}_0$, ($\varepsilon^t_{\omega_0^2}$). Debido a esta multiplicación se necesitan tres por cada iteración para el cálculo de la raíz cuadrada recíproca.

La expresión de K para la raíz cuadrada, teniendo en cuenta los errores, es la siguiente:

$$\hat{K}_1 = \frac{1}{2}(3 - \hat{r}_0) + \varepsilon^t_{K_1} \tag{2.23}$$

Reemplazando (2.22) en (1.48) se obtiene:

$$\hat{K}_1 = 1 - \frac{\varepsilon_0^2}{2\omega^2} - \frac{\varepsilon_0}{\omega} - \frac{\varepsilon^t_{\omega_0^2}}{2\omega^2} - \frac{\varepsilon^t_{r_0}}{2} + \varepsilon^t_{K_1} \tag{2.24}$$

Una vez que se dispone de $\hat{K}_1$, se puede calcular $\hat{\omega}_1$ mediante .

$$\hat{\omega}_1 = \hat{\omega}_0 \cdot \hat{K}_1 + \varepsilon^t_{\omega_1} \tag{2.25}$$

Esta ecuación se expande introduciendo en ella el valor de K_1 de la ecuación (2.24)

$$\hat{\omega}_1 = (\omega + \varepsilon_0) \cdot [1 - \frac{\varepsilon_0^2}{2\omega^2} - \frac{\varepsilon_0}{\omega} - \frac{\varepsilon^t_{\omega_0^2}}{2\omega^2} - \frac{\varepsilon^t_{r_0}}{2} + \varepsilon^t_{K_1}] + \varepsilon^t_{\omega_1} \tag{2.26}$$

La ecuación anterior sirve para obtener la expresión del error del resultado para la primera iteración.

$$\begin{aligned} \varepsilon_1 \quad = \quad & -\frac{3}{2\omega} \cdot \varepsilon_0^2 - \frac{1}{2\omega^2} \cdot \varepsilon_0^3 - \frac{1}{2\omega^2} \cdot \varepsilon^t_{\omega_0^2} - \omega \cdot (\frac{\varepsilon^t_{r_0}}{2} - \varepsilon^t_{K_1}) \\ & -\varepsilon_0 \cdot (\frac{\varepsilon^t_{r_0}}{2} - \varepsilon^t_{K_1}) - \frac{1}{2\omega^2} \cdot \varepsilon_0 \cdot \varepsilon^t_{\omega_0^2} + \varepsilon^t_{\omega_1} \end{aligned} \tag{2.27}$$

Siguiendo los mismos pasos que para el caso del recíproco, es posible obtener la expresión del error del resultado para una iteración genérica i.

$$\begin{aligned} \varepsilon_i \quad = \quad & -\frac{3}{2\omega} \cdot \varepsilon_{i-1}^2 - \frac{1}{2\omega^2} \cdot \varepsilon_{i-1}^3 - \frac{1}{2\omega^2} \cdot \varepsilon^t_{\omega^2_{i-1}} - \omega \cdot (\frac{\varepsilon^t_{r_{i-1}}}{2} - \varepsilon^t_{K_i}) \\ & -\varepsilon_{i-1} \cdot (\frac{\varepsilon^t_{r_{i-1}}}{2} - \varepsilon^t_{K_i}) - \frac{1}{2\omega^2} \cdot \varepsilon_{i-1} \cdot \varepsilon^t_{\omega^2_{i-1}} + \varepsilon^t_{\omega_i} \end{aligned} \tag{2.28}$$

Los dos primeros términos de esta expresión se corresponden con el error de aproximación del algoritmo. El resto de los términos depende del error de las operaciones intermedias. Estos términos pueden hacer que el error total sea positivo en algunos casos.

2.3. Límites del error

Las ecuaciones obtenidas en la sección previa van a ser usadas aquí para poder establecer unos límites (superior e inferior) para el error del resultado en una iteración determinada del algoritmo. Estos límites serán usados para obtener el tamaño de palabra óptimo para las operaciones intermedias a partir de un requerimiento de exactitud para el resultado. Los requerimientos de exactitud siempre se expresan como desigualdades. Por eso, se utilizan los límites

para obtener el tamaño de palabra óptimo para las operaciones intermedias. Antes de comenzar con este cálculo, es necesario establecer algunas suposiciones necesarias acerca de la arquitectura de las unidades hardware para llevarlo a cabo.

Ya se ha mencionado en ocasiones anteriores que todos los algoritmos multiplicativos para división, raíz cuadrada y sus recíprocos comienzan con la obtención de una aproximación al resultado (semilla). Esta aproximación se obtiene de la lectura de tablas y a continuación se realiza una serie de iteraciones. En los diseños más comunes siempre se utiliza una semilla con una precisión tal que no se requieren más de una o dos iteraciones para obtener el resultado con una exactitud adecuada [Obe99]. De manera que se considerarán los casos de obtención del resultado con una o dos iteraciones.

Si suponemos que se obtiene una aproximación inicial que tiene b bits fraccionales, tendrá un error máximo de 1 *ulp* ($\pm 2^{-b}$). Se asumirá que todas las operaciones intermedias en cada iteración calculan sus resultados con el mismo número de bits fraccionales. Desde el punto de vista del diseño hardware, esto hace posible que se puedan implementar estos algoritmos mediantes el reuso del multiplicador existente en la arquitectura [ES00]. El método de redondeo considerado en las operaciones intermedias para este propósito será el redondeo al más próximo por ser el más común. Para este método de redondeo el error del resultado de las operaciones intermedias es 0.5 *ulp*. El mismo proceso que se va a describir aquí podría ser llevado a cabo considerando cualquiera de los otros métodos de redondeo propuestos por el estándar IEEE 754.

Teniendo en cuenta que los resultados de las multiplicaciones se proporcionan con p bits fraccionales y que el método de redondeo es al más próximo, el error máximo producido por una multiplicación será $\pm 2^{-p-1}$. De este modo se relacionan el error con el tamaño del multiplicador. Lo que se busca obtener la expresión del error total para una iteración en función del tamaño del multiplicador. Así, teniendo el error como requerimiento será posible obtener el tamaño óptimo del multiplicador. Entonces, el objetivo será obtener los valores máximo y mínimo del error estableciendo límites para cada término. Una vez obtenidos estos límites, haciéndolos coincidir con los requerimientos de exactitud es posible obtener el tamaño de palabra óptimo para las operaciones intermedias. Este es el objetivo de este capítulo.

2.3.1. Recíproco

Considerando dos iteraciones, tal y como se ha hecho hasta ahora, seguimos denotando la acción de leer la semilla como iteración 0. De lo que se trata es

de encontrar los valores máximo y mínimo de la ecuación (2.19). Para conseguir esto es necesario encontrar los valores máximo y mínimo de cada uno de sus términos.

Una iteración: En primer lugar, es necesario reescribir la ecuación (2.19) del error para la primera iteración.

$$\varepsilon_1 = -\frac{1}{\omega}\varepsilon_0^2 - \omega \cdot (\varepsilon_{r_0}^t - \varepsilon_{K_1}^t) - \varepsilon_0 \cdot (\varepsilon_{r_0}^t - \varepsilon_{K_1}^t) + \varepsilon_{\omega_1}^t \tag{2.29}$$

De acuerdo con lo dicho anteriormente acerca de las operaciones intermedias podemos establecer los límites para cada una de las contribuciones al error que tenemos en esta ecuación.

$$\begin{array}{llll} -2^{-b} & \leq \varepsilon_0 \leq & 2^{-b} \\ -2^{-p-1} & \leq \varepsilon_{r_0}^t \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon_{K_1}^t \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon_{\omega_1}^t \leq & 2^{-p-1} \end{array} \tag{2.30}$$

El primero de ellos es el error de la aproximación inicial, el segundo es el error de la multiplicación del cálculo de r_0, el tercero es el error del complemento a dos para calcular K_1 y el tercero la multiplicación que interviene en el cálculo del resultado de la primera iteración (ω_1).

Teniendo en cuenta el primero de los límites en (2.30) nos permite obtener los límite para el primer término de (2.29).

$$-2^{-2\cdot b+1} \leq -\frac{1}{\omega}\varepsilon_0^2 \leq 0 \tag{2.31}$$

Este término nunca es positivo. Esto es conocido a priori dado que es el error de aproximación del algoritmo. Para el siguiente de los términos es necesario tener en cuenta que los operandos están siempre en el intervalo $[1, 2)$. Entonces, el resultado del recíproco ω estará en el intervalo $(0{,}5, 1]$. Sumando los errores máximos de $\varepsilon_{r_0}^t$, $\varepsilon_{K_1}^t$ y multiplicando por el valor máximo de ω se obtienen los límites para el segundo término:

$$-2^{-p} \leq \omega(\varepsilon_{r_0}^t - \varepsilon_{K_1}^t) \leq 2^{-p} \tag{2.32}$$

En el caso del tercer término, los límites involucran también el resultado de la aproximación inicial.

$$-2^{-b} \cdot 2^{-p} \leq \varepsilon_0 \cdot (\varepsilon_{r_0}^t - \varepsilon_{K_1}^t) \leq 2^{-b} \cdot 2^{-p} \tag{2.33}$$

El último de los términos es, simplemente, el error de la multiplicación necesaria para obtener el resultado de la primera iteración ($\hat{\omega}_1$). Sus límites son:

$$-2^{-p-1} \leq \varepsilon^t_{\omega_1} \leq 2^{-p-1} \tag{2.34}$$

Agrupando todos los límites superiores y todos los inferiores, obtenemos los límites globales para el error de la aproximación al resultado para la primera iteración.

$$-2^{-2b+1} - 2^{-p-1} \cdot (3 + 2^{-b+1}) \leq \varepsilon_1 \leq 2^{-p-1} \cdot (3 + 2^{-b+1}) \tag{2.35}$$

Dos iteraciones: El proceso para obtener los límites para el caso de dos iteraciones es muy similar. Se parte de la expresión del error de la aproximación al resultado para la segunda iteración.

$$\varepsilon_2 = -\frac{1}{\omega}\varepsilon_1^2 - \omega \cdot (\varepsilon^t_{r_1} - \varepsilon^t_{K_2}) - \varepsilon_1 \cdot (\varepsilon^t_{r_1} - \varepsilon^t_{K_2}) + \varepsilon^t_{\omega_2} \tag{2.36}$$

Excepto el error de la iteración anterior (ε_1), todas las contribuciones al error son parecidas al caso anterior. Son las siguientes:

$$\begin{array}{rrl} -2^{-2b+1} - 2^{-p-1} \cdot (3 + 2^{-b+1}) & \leq \varepsilon_1 \leq & 2^{-p-1} \cdot (3 + 2^{-b+1}) \\ -2^{-p-1} & \leq \varepsilon^t_{r_1} \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon^t_{K_2} \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon^t_{\omega_1} \leq & 2^{-p-1} \end{array} \tag{2.37}$$

El siguiente paso a seguir es ir obteniendo los límites para cada uno de los términos de la ecuación (2.36). Para el primer término hay que tener en cuenta el valor de los límites para (ε_1) establecidos en (2.37).

$$\begin{array}{rcl} -\frac{1}{\omega}\varepsilon_1^2 & \geq & -2 \cdot [2^{-2b+1} + 2^{-p-1} \cdot (3 + 2^{-b+1})]^2 \\ -\frac{1}{\omega}\varepsilon_1^2 & \leq & 0 \end{array} \tag{2.38}$$

Estas expresiones pueden ser simplificadas evitando así tener unas expresiones finales muy complicadas y que sean igualmente válidas. En la figura 2.1 se puede ver la representación del término $3 + 2^{-b+1}$ que aparece en los límites

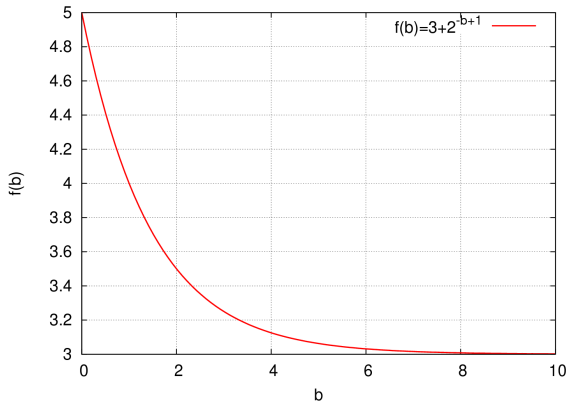


Figura 2.1: Función $f(b) = 3 + 2^{-b+1}$

de la ecuación (2.38). El tamaño de la aproximación inicial (b) es, para dos iteraciones, de 7 bits. Si el esquema fuese de una sola iteración la aproximación tendría aún más bits. Para ese valor, se puede ver en la gráfica que, para $b = 7$, ese término es prácticamente 3 y el término puede ser sustituido por ese valor.

Aplicando la aproximación se obtiene:

$$-2 \cdot [2^{-2b+1} + 2^{-p-1} \cdot (3 + 2^{-b+1})]^2 \simeq -2 \cdot (2^{-2b+1} + 3 \cdot 2^{-p-1})^2 \tag{2.39}$$

Si se expande la expresión aproximada:

$$\simeq -2 \cdot (2^{-4b+2} + 2^{-2p-2} + 3 \cdot 2^{-p} \cdot 2^{-2b+1}) \tag{2.40}$$

Dados que $2p$ es el doble del ancho con el que se realizan las operaciones intermedias, el término $3 \cdot 2^{-2p-2}$ es claramente despreciable. Integrando todas estas simplificaciones, los límites para este término son:

$$\begin{aligned} -\frac{1}{\omega}\varepsilon_1^2 &\leq 0 \\ -\frac{1}{\omega}\varepsilon_1^2 &\geq -2^{-4b+3} - 3 \cdot 2^{-2b+2} \cdot 2^{-p} \end{aligned} \tag{2.41}$$

Los límites para el segundo término de la ecuación (2.36) se pueden extraer fácilmente con los mismos argumentos que para la iteración anterior.

$$-2^{-p} \leq \omega \cdot (\varepsilon_{r_1}^t - \varepsilon_{K_2}^t) \leq 2^{-p} \tag{2.42}$$

En el caso del tercer término hay que sustituir en él los límites del error de la iteración anterior (ecuación (2.35)). Haciendo esto se obtiene:

$$\begin{aligned} \varepsilon_1 \cdot (\varepsilon_{r_1}^t - \varepsilon_{K_2}^t) &\geq -2^{-p} \cdot [2^{-2b+1} + 2^{-p-1} \cdot (3 + 2^{-b+1})] \\ \varepsilon_1 \cdot (\varepsilon_{r_1}^t - \varepsilon_{K_2}^t) &\leq 2^{-p} \cdot [2^{-2b+1} + 2^{-p-1} \cdot (3 + 2^{-b+1})] \end{aligned} \tag{2.43}$$

Para este término también es posible hacer una aproximación. Al expandir la expresión aparecen términos multiplicados por 2^{-2p}. Estos términos se explicó anteriormente que son despreciables. Por lo tanto, la expresión final para los límites del tercer término es:

$$\begin{aligned} \varepsilon_1 \cdot (\varepsilon_{r_1}^t - \varepsilon_{K_2}^t) &\geq -2^{-p} \cdot 2^{-2b+1} \\ \varepsilon_1 \cdot (\varepsilon_{r_1}^t - \varepsilon_{K_2}^t) &\leq 2^{-p} \cdot 2^{-2b+1} \end{aligned} \tag{2.44}$$

Los límites del último término son iguales que en la iteración anterior.

$$-2^{-p-1} \leq \varepsilon_{\omega_2}^t \leq 2^{-p-1} \tag{2.45}$$

Agrupando todos los límites superiores y todos los límites inferiores (sin ninguna aproximación) obtenemos la expresión de los límites para la segunda iteración.

$$-2^{-4b+3} - 2^{-p-1} \cdot (3 + 7 \cdot 2^{-2b+2}) \leq \varepsilon_2 \leq 2^{-p-1} \cdot (3 + 2^{-2b+2}) \tag{2.46}$$

Estas expresiones hacen posible el cálculo del tamaño de las operaciones intermedias (p) para un tamaño aproximación inicial concreta (b). El error final del resultado, es decir la exactitud con que ha de calcularse el resultado, es un requerimiento del diseño. Por lo tanto, es dado a a priori. Al igual que para el caso de la expresión del error del resultado para cada iteración, estos límites han sido validados por medio de simulaciones exhaustivas, como se verá más adelante.

2.3.2. Raíz cuadrada recíproca

Realizamos ahora la obtención de los límites del error de la aproximación al resultado para la raíz cuadrada recíproca. Se vuelven a proponer los límites para una y dos iteraciones. Se trata, ahora, de obtener los valores máximo y mínimo de la ecuación (2.28).

Una iteración: La expresión del error para este esquema es como sigue:

$$\begin{aligned}\varepsilon_1 &= -\frac{3}{2\omega}\cdot\varepsilon_0^2 - \frac{1}{2\omega^2}\cdot\varepsilon_0^3 - \frac{1}{2\omega^2}\cdot\varepsilon^t_{\omega_0^2} - \omega\cdot(\frac{\varepsilon^t_{r_0}}{2} - \varepsilon^t_{K_1}) \\ &\quad -\varepsilon_0\cdot(\frac{\varepsilon^t_{r_0}}{2} - \varepsilon^t_{K_1}) - \frac{1}{2\omega^2}\cdot\varepsilon_0\cdot\varepsilon^t_{\omega_0^2} + \varepsilon^t_{\omega_1}\end{aligned} \tag{2.47}$$

Las contribuciones al error de cada una de las operaciones intermedias son prácticamente las misma que las de (2.30). Esto es así porque las operaciones intermedias siguen siendo las mismas. Hay una contribución adicional puesto que, en cada iteración para la raíz cuadrada recíproca, se calcula el cuadrado de la aproximación al resultado en la iteración anterior. Dicha contribución al error viene dada por:

$$-2^{-p-1} \leq \varepsilon^t_{\omega_0^2} \leq 2^{-p-1} \tag{2.48}$$

La diferencia viene dada por la evaluación de cada una de los términos de la expresión del error. El primero de los términos tiene los siguientes límites.

$$-3\cdot 2^{-2b} \leq -\frac{3}{2\omega}\varepsilon_0^2 \leq 0 \tag{2.49}$$

Con el segundo término completamos la parte correspondiente al error de aproximación. Sus límites son:

$$-2^{-3b+1} \leq -\frac{1}{2\omega^2}\varepsilon_0^3 \leq 2^{-3b+1} \tag{2.50}$$

A partir de aquí aparecen los términos que involucran los errores de las operaciones intermedias. El primero de ellos es el tercer término de la expresión para el error total.

$$-2^{-p} \leq \frac{1}{2\omega^2}\cdot\varepsilon^t_{\omega_0^2} \leq 2^{-p} \tag{2.51}$$

El cuarto término es:

$$-\frac{3}{2}\cdot 2^{-p-1} \leq \omega\cdot(\frac{\varepsilon_{r_0}^t}{2}-\varepsilon_{K_1}^t) \leq \frac{3}{2}\cdot 2^{-p-1} \tag{2.52}$$

El quinto término es similar pero involucra el error total de la iteración anterior.

$$-\frac{3}{2}\cdot 2^{-p-1}\cdot 2^{-b} \leq \varepsilon_0\cdot(\varepsilon_{r_0}^t-\varepsilon_{K_1}^t) \leq \frac{3}{2}\cdot 2^{-p-1}\cdot 2^{-b} \tag{2.53}$$

El sexto término tiene los siguientes límites:

$$-2^{-b+1}\cdot 2^{-p-1} \leq -\frac{1}{2\omega^2}\cdot\varepsilon_0\cdot\varepsilon_{\omega_0^2}^t \leq 2^{-b+1}\cdot 2^{-p-1} \tag{2.54}$$

El último término es, otra vez, el error de la multiplicación que se realiza para obtener el resultado.

$$-2^{-p-1} \leq \varepsilon_{\omega_1}^t \leq 2^{-p-1} \tag{2.55}$$

Para calcular los límites del error total hay que agrupar los términos. En el caso del límite inferior se hace una aproximación basada en la figura 2.2. Consiste en una gráfica en la que se comparan los valores de los límites inferiores del primer y segundo término del error para tamaños típicos de la aproximación inicial. A la vista de la gráfica se concluye que la contribución del error del segundo término es despreciable frente a la primera

Teniendo en cuenta la aproximación citada y el resto de las contribuciones al error los límites son:

$$\begin{aligned}\varepsilon_1 &\leq 2^{-3b+1}+\frac{9}{2}\cdot 2^{-p-1}+\frac{7}{2}\cdot 2^{-p-1}\cdot 2^{-b}\\ \varepsilon_1 &\geq -3\cdot 2^{-2b}-\frac{9}{2}\cdot 2^{-p-1}-\frac{7}{2}\cdot 2^{-p-1}\cdot 2^{-b}\end{aligned} \tag{2.56}$$

Dos iteraciones: Queda por tratar el esquema con dos iteraciones. El error del resultado en la segunda iteración es:

$$\begin{aligned}\varepsilon_2 = &-\frac{3}{2\omega}\cdot\varepsilon_1^2-\frac{1}{2\omega^2}\cdot\varepsilon_1^3-\frac{1}{2\omega^2}\cdot\varepsilon_{\omega_1^2}^t-\omega\cdot(\frac{\varepsilon_{r_1}^t}{2}-\varepsilon_{K_2}^t)\\ &-\varepsilon_1\cdot(\frac{\varepsilon_{r_1}^t}{2}-\varepsilon_{K_2}^t)-\frac{1}{2\omega^2}\cdot\varepsilon_1\cdot\varepsilon_{\omega_1^2}^t+\varepsilon_{\omega_2}^t\end{aligned} \tag{2.57}$$

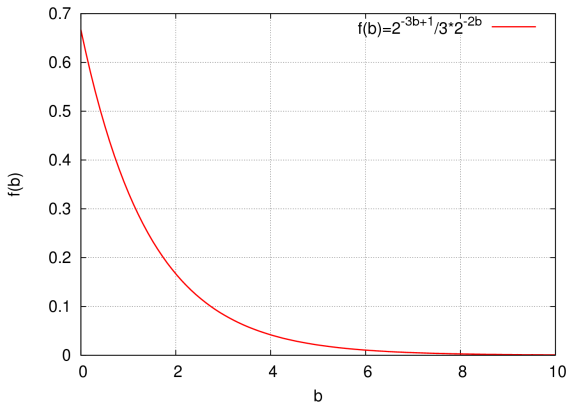


Figura 2.2: Función $f(b) = 2^{-3b+1}/3 * 2^{-2b}$

Para este caso las contribuciones individuales de las operaciones intermedias son completamente análogas a los casos anteriores, así que ya no son explicitadas aquí. Los límites del primer término de la expresión del error se puede obtener fácilmente. Para este caso únicamente se deprecian, igual que se hizo en otros casos, aquellos términos que son múltiplos de 2^{-2p} por ser muy pequeños. Para hacer las expresiones más manejables, únicamente se hace una aproximación en el límite inferior, cuya expresión es:

$$-3 \cdot [9 \cdot 2^{-4b} + 2^{-p-1} \cdot (27 \cdot 2^{-2b} + 21 \cdot 2^{-3b})] \tag{2.58}$$

La gráfica 2.3 muestra que este la expresión anterior puede ser aproximada por:

$$-3 \cdot [9 \cdot 2^{-4b} + 27 \cdot 2^{-2b} \cdot 2^{-p-1}] \tag{2.59}$$

Por lo tanto los límites para el primer término son:

$$-27 \cdot 2^{-4b} + 81 \cdot 2^{-2b} \cdot 2^{-p-1} \leq \frac{-3}{2\omega}\varepsilon_1^2 \leq 0 \tag{2.60}$$

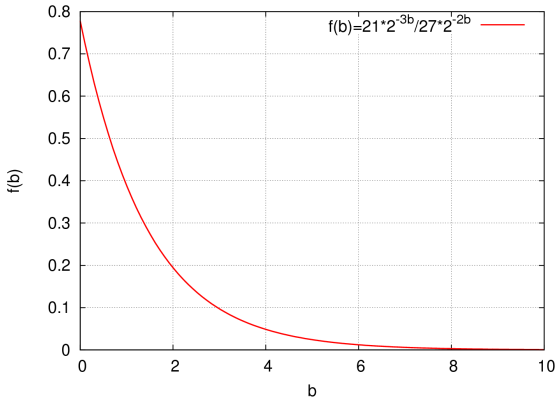


Figura 2.3: Función $f(b) = 21 * 2^{-3b}/27 * 2^{-2b}$

En el caso del segundo término partimos del cuadrado del error de la iteración anterior obtenido en el análisis del anterior término para obtener el cubo de dicho error. Igualmente se desprecian todos aquellos términos que son múltiplos de 2^{-2p}. En valor absoluto, el valor máximo que este término puede tomar es:

$$27 \cdot 2^{-6b} + 2^{-p-1} \cdot (\frac{243}{2} \cdot 2^{-4b} + \frac{187}{2} \cdot 2^{-5b}) \tag{2.61}$$

A la vista de la gráfica 2.4 se puede ver que la ecuación anterior puede ser simplificada. Una vez hecho esto queda de la siguiente forma:

$$27 \cdot 2^{-6b} + \frac{243}{2} \cdot 2^{-4b} \cdot 2^{-p-1} \tag{2.62}$$

Los límite para este término:

$$-27 \cdot 2^{-6b+1} + 243 \cdot 2^{-4b} \cdot 2^{-p-1} \leq \frac{-1}{2\omega^2} \cdot \varepsilon_1^3 \leq 27 \cdot 2^{-6b+1} + 243 \cdot 2^{-4b} \cdot 2^{-p-1} \tag{2.63}$$

El tercer término tiene los valores límite:

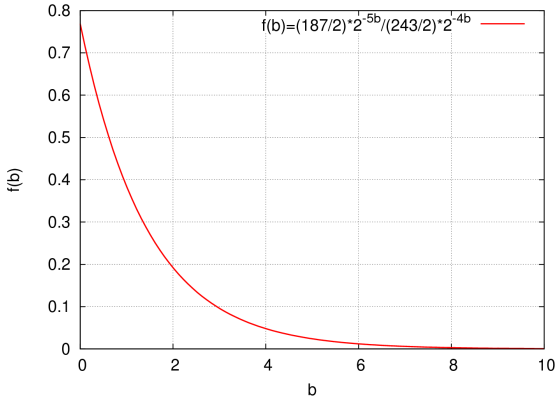


Figura 2.4: Función $f(b) = (187/2) * 2^{-5b}/(243/2) * 2^{-4b}$

$$-2^{-p} \leq \frac{1}{2\omega^2} \cdot \varepsilon^t_{\omega_0^2} \leq 2^{-p} \tag{2.64}$$

Para el cuarto término los límites son:

$$-\frac{3}{2} \cdot 2^{-p-1} \leq \omega \cdot (\frac{\varepsilon^t_{r_0}}{2} - \varepsilon^t_{K_1}) \leq \frac{3}{2} \cdot 2^{-p-1} \tag{2.65}$$

En el cálculo de los límites del quinto término, lo único que hay que tener en cuenta es que se desprecian los términos que son múltiplos 2^{-2p}.

$$-\frac{9}{2} \cdot 2^{-2b} \cdot 2^{-p-1} \leq \varepsilon_0(\frac{\varepsilon^t_{r_0}}{2} - \varepsilon^t_{K_1}) \leq \frac{9}{2} \cdot 2^{-2b} \cdot 2^{-p-1} \tag{2.66}$$

Con los mismos criterios que en el caso anterior se obtienen los límites para el sexto término.

$$-6 \cdot 2^{-2b} \cdot 2^{-p-1} \leq \frac{-1}{2\omega^2} \cdot \varepsilon_1 \cdot \varepsilon^t_{\omega_1^2} \leq 6 \cdot 2^{-2b} \cdot 2^{-p-1} \tag{2.67}$$

Para el último de los términos:

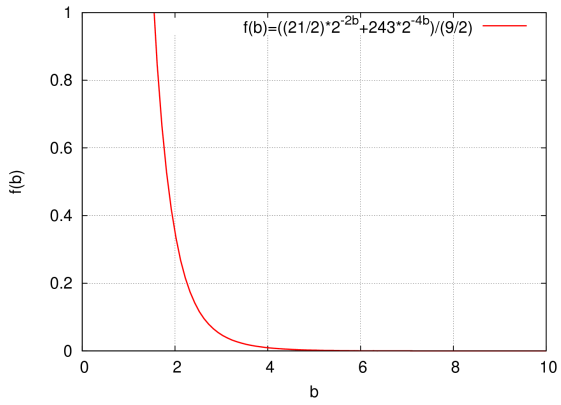


Figura 2.5: Función $f(b) = ((21/2) * 2^{-2b} + 243 * 2^{-4b})/(9/2)$

$$-2^{-p-1} \leq \varepsilon^{t}_{\omega_1} \leq 2^{-p-1} \tag{2.68}$$

La expresión final de los límites se obtiene mediante la agrupación de los límites superiores e inferiores. Comenzando por los superiores, la agrupación de los valores máximos produce la siguiente expresión:

$$27 \cdot 2^{-6b+1} + 2^{-p-1} \cdot (\frac{9}{2} + \frac{21}{2} \cdot 2^{-2b} + 243 \cdot 2^{-4b}) \tag{2.69}$$

Se puede realizar una aproximación en los términos que están multiplicados por 2^{-p-1}. Esta aproximación está argumentada en la gráfica 2.5. De manera que la ecuación (2.69) se aproxima por:

$$27 \cdot 2^{-6b+1} + \frac{9}{2} \cdot 2^{-p-1} \tag{2.70}$$

En el caso de los límites inferiores, la agrupación de los límites produce el valor mínimo:

$$-27 \cdot 2^{-4b} - 27 \cdot 2^{-6b+1} - 2^{-p-1} \cdot (\frac{9}{2} + \frac{183}{2} \cdot 2^{-2b} + 243 \cdot 2^{-4b}) \tag{2.71}$$

En este caso, para producir una expresión más compacta vamos a aplicar dos aproximaciones, según las gráficas 2.6b y 2.6a.

Así, finalmente, los límites del error del resultado para un esquema de dos iteraciones es como sigue (se realiza una aproximación adicional para poder trabajar con potencias de 2):

$$-2^{-4b+5} + \frac{9}{2} \cdot 2^{-p-1} \leq \varepsilon_2 \leq 2^{-6b+6} + \frac{9}{2} \cdot 2^{-p-1} \tag{2.72}$$

Al igual que en el caso del recíproco estas expresiones pueden ser utilizadas para la obtención del tamaño óptimo del multiplicador para un requerimiento del error y una tamaño de aproximación inicial dadas.

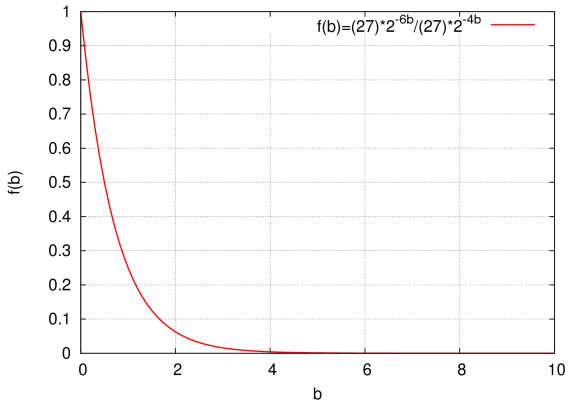
Disponemos en este punto de las expresiones del error del resultado en cualquier iteración para las distintas operaciones consideradas. El resultado es poder conocer de manera analítica el error del resultado. Se han obtenido también los límites que los valores del error del resultado puede tomar. Estos límites podrán ser utilizados para poder calcular el tamaño de palabra óptimo para las operaciones intermedias. Se presenta, a continuación, la validación de todas las expresiones obtenidas.

2.4. Simulaciones

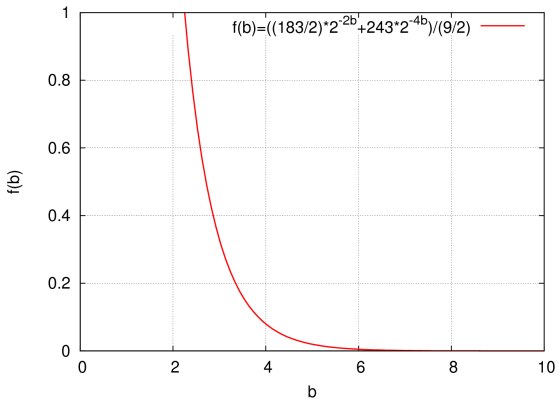
Para validar las expresiones del error y sus límites, obtenidos en la sección anterior, se ha utilizado la simulación. Lo que se ha simulado es la ejecución hardware del algoritmo GLD. La herramienta utilizada fue Maple [GLM08]. Esta simulación contabiliza el efecto que el hardware y el redondeo tiene en los resultados de las operaciones aritméticas.

Se realizaron simulaciones exhaustivas, lo que quiere decir que han sido simulados todos los valores posibles de los operandos. La precisión utilizada en estas simulaciones fue la precisión simple del estándar IEEE [IEE08] (23 bits de mantisa, 8 bits de exponente y 1 bit de signo). Este formato es muy común en la aritmética de computadores. Adicionalmente, permite realizar las simulaciones en un tiempo razonable de tiempo. Las simulaciones para doble precisión serían demasiado costosas computacionalmente.

Las arquitecturas simuladas serán las mismas que las utilizadas en la sección anterior. Se realiza la lectura de la aproximación de una tabla. Se consideran



(a) Función $f(b) = (27) * 2^{-6b}/(27) * 2^{-4b}$



(b) Función $f(b) = ((183/2) * 2^{-2b} + 243 * 2^{-4b})/(9/2)$

Figura 2.6: Aproximaciones para el límite inferior en el esquema de dos iteraciones

dos iteraciones de modo que así es posible validar todas la expresiones para una y dos iteraciones.

El código diseñado para la simulación tiene dos partes diferenciadas. En la primera se evalúan de manera teórica la expresiones obtenidas para el error en las respectivas iteraciones. La segunda parte simula la ejecución hardware del algoritmo teniendo en cuenta el efecto del error en las operaciones intermedias. Cada operación intermedia añade un error. El redondeo implementado en las multiplicaciones es el redondeo al más próximo.

Para todas las simulaciones los parámetros importantes que hay que ajustar son el número de bits decimales de la aproximación inicial (b) y el número de bits decimales en los resultados de las operaciones intermedias (p). En todos los casos, el tamaño de la aproximación inicial b se fija a 7 bits (suficiente para alcanzar la exactitud exigida en dos iteraciones). El tamaño de palabra de las operaciones intermedias p se fija a 26 (algunos bits extra a mayores de la precisión del formato, que es 23). Estos son los bits extra que se usan para llevar a cabo el redondeo del resultado.

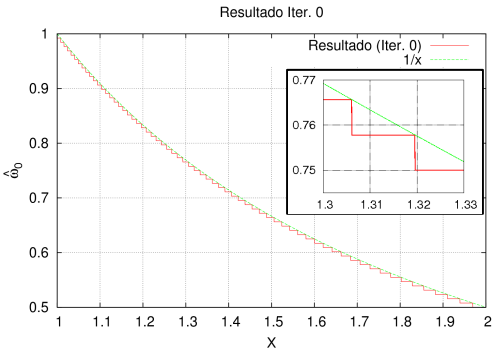
El error de cada iteración se mide como la diferencia entre los resultados obtenidos y el resultado exacto para cada iteración. Como se ha calculado de manera teórica los errores de los resultados, es posible comparar los resultados teóricos y los medidos.

2.4.1. Recíproco

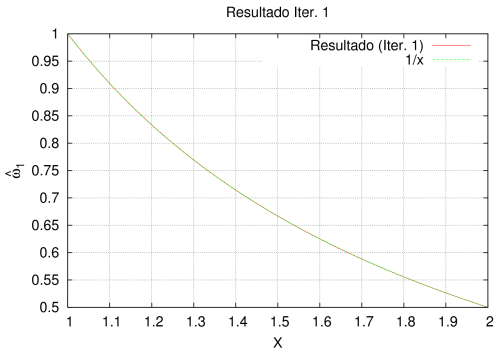
En este apartado se presentan los resultados de las simulaciones para el recíproco. El primer aspecto que es necesario corroborar es que los resultados que se obtienen del algoritmo son los correctos. Es decir, que el algoritmo converge adecuadamente hacia el resultado.

En las figuras 2.7a, 2.7b y 2.7c están representados los resultados para todos los posibles operandos de cada una de las iteraciones. En la primera de estas figuras se pueden observar los distintos valores que toma la aproximación inicial al resultado. En las gráficas, ya a partir de la primera iteración, apenas se distingue la gráfica del resultado exacto del resultado calculado por el algoritmo. Las consideraciones concretas del error se harán a continuación pero se ve claramente que el resultado obtenido converge al deseado. Por lo tanto, el primer objetivo, que era comprobar si se estaba calculando los resultados de manera adecuada, queda satisfecho.

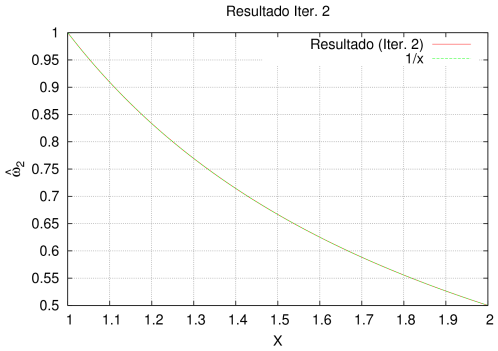
El siguiente aspecto que se comprobó con las simulaciones es muy importante. Se trata de ver si el error de la aproximación al resultado producido por el algoritmo es correctamente predicho por la expresiones obtenidas para tal fin.



(a) Resultado del recíproco para la Iteración 0



(b) Resultado del recíproco para la Iteración 1



(c) Resultado del recíproco para la Iteración 2

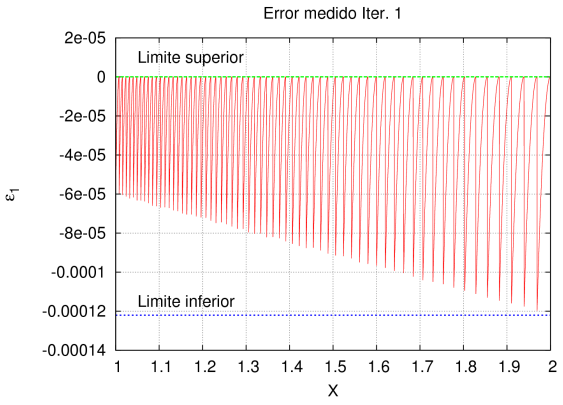
Figura 2.7: Resultado del recíproco en las distintas iteraciones para el algoritmo GLD

Los casos relevantes son ahora la primera y segunda iteración del algoritmo. Además, se comprueba también el tercer aspecto. Los límites obtenidos mediante aproximaciones son correctos y predicen de manera adecuada los valores máximo y mínimo del error. Esto se hace dando valores a las expresiones de los límites, utilizando los característicos de las simulaciones definidos antes, $b = 7$ y $p = 23$.

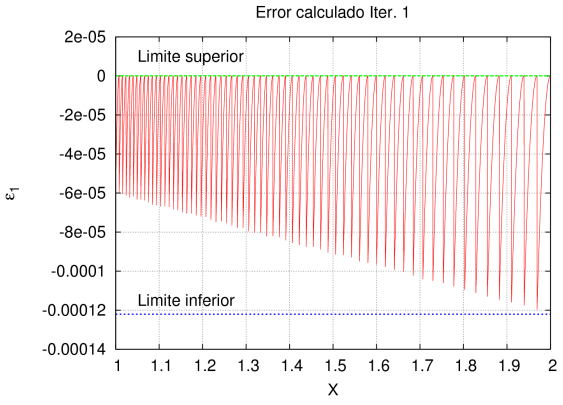
En la figura 2.8 podemos ver el error de la aproximación al resultado para la primera iteración. Como se puede ver, el error medido y el error obtenido de las ecuaciones del modelo coincide exactamente. Aparecen algunos, aunque muy pocos, puntos con valores del error positivo. Como habíamos dicho esto es debido a la representación finita de los números en hardware. Con todo el error se mantiene negativo para la mayoría de los puntos. También se puede ver que el valor mínimo del error depende de X. Esto es lógico puesto que todavía predomina la parte del error de aproximación ($-\frac{1}{\omega} \cdot \varepsilon_0^2$) en la ecuación 2.19. Se pueden ver, además representados los valores de los límites para la primera iteración en ambas gráficas. Estos valores son obtenidos sustituyendo $p = 23$ y $b = 7$ en la ecuación 2.35. Como se puede ver, estos límites han sido adecuadamente calculados.

La figura 2.9 contiene las gráficas del error de la aproximación al resultado correspondientes a la segunda iteración. Al igual que para la iteración anterior, tanto el error medido como el calculado de manera teórica coinciden. En este caso todavía se aprecia, pero de manera más suave, la contribución del error de aproximación. Sin embargo, ahora tiene un fuerte predominio la parte del error debido a la contribución de los errores de las operaciones intermedias. Por eso, ahora el número de puntos con error positivo es mucho mayor que en el caso anterior. Esto es así porque, en estos casos, el peso del error de las operaciones intermedias es comparable o mayor que el error de aproximación del algoritmo. Al igual que en el caso anterior, también se incluye en ambas gráficas los valores de los límites para la segunda iteración sustituyendo el tamaño de las operaciones intermedias y de la aproximación inicial ($p = 23$ y $b = 7$) en la ecuación 2.46. Una vez más se puede ver claramente en la gráfica que los límites predicen de manera adecuada el rango de valores que puede tomar el error de la aproximación al resultado en la segunda iteración.

Las figuras 2.10a, 2.10b y 2.10c representan el error de cada una de las tres iteraciones pero en escala logarítmica. Lo que se representa en el eje de ordenadas es el logaritmo en base dos del error de la aproximación al resultado para una iteración concreta. De este modo, se puede observar de manera inmediata a que posición de bit fraccional al que afecta el error.

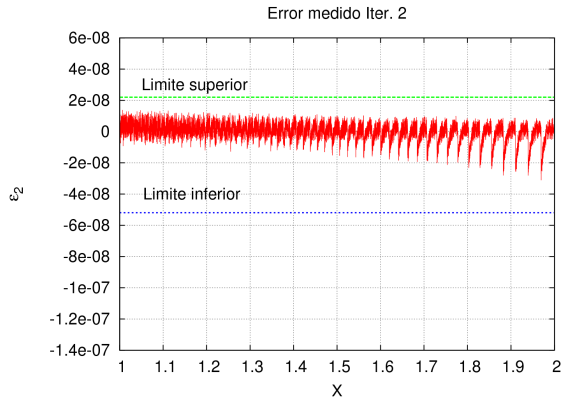


(a) Error medido en la primera iteración

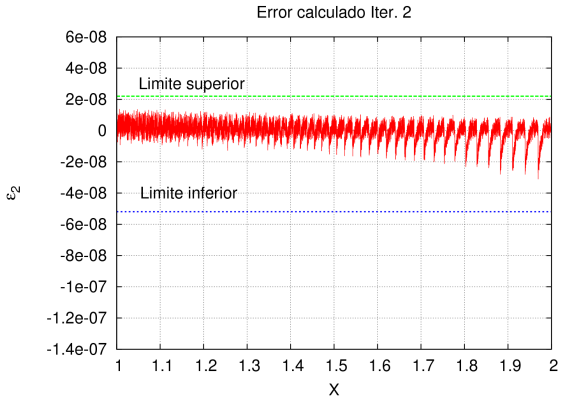


(b) Error en la primera iteración obtenido de las expresiones teóricas

Figura 2.8: Errores de la aproximación al resultado para la primera iteración del recíproco

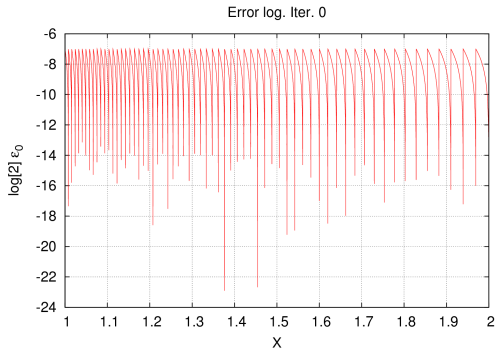


(a) Error medido en la segunda iteración

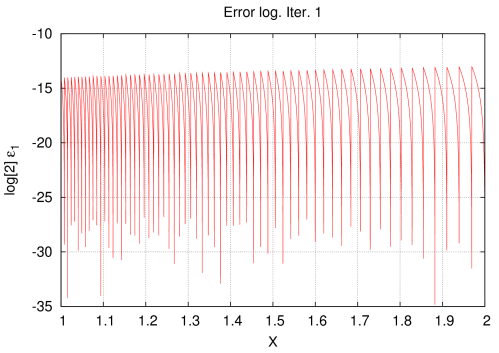


(b) Error en la segunda iteración obtenido de las expresiones teóricas

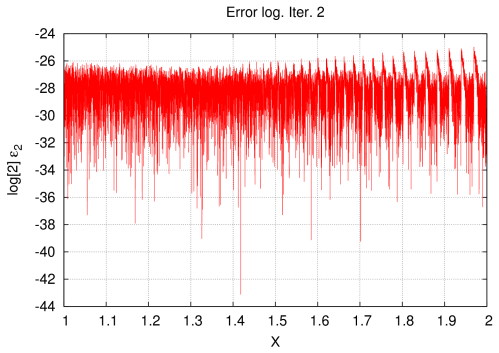
Figura 2.9: Errores de la aproximación al resultado para la segunda iteración del recíproco



(a) Error logarítmico para la Iteración 0 del recíproco



(b) Error logarítmico para la Iteración 1 del recíproco



(c) Error logarítmico para la Iteración 2 del recíproco

Figura 2.10: Error logarítmico del recíproco en las distintas iteraciones para el algoritmo de Goldschmidt

2.4.2. Raíz cuadrada recíproca

Tal y como se mencionó al principio de la sección también se realizaron simulaciones para la raíz cuadrada recíproca. Los parámetros utilizados para la simulación son los mismos que para el recíproco ($p = 23$ y $b = 7$). En este apartado se presentan los resultados. La primera de las comprobaciones fue la convergencia del algoritmo hacia el resultado adecuado. Al igual que en el caso del recíproco se presentan los resultados para la aproximación inicial y las dos iteraciones en las figuras 2.11a, 2.11b y 2.11b. En la primera de estas gráficas se puede observar claramente la diferencia entre el resultado exacto y la aproximación inicial utilizada. Las siguientes demuestran como el resultado converge rápidamente hacia el resultado deseado.

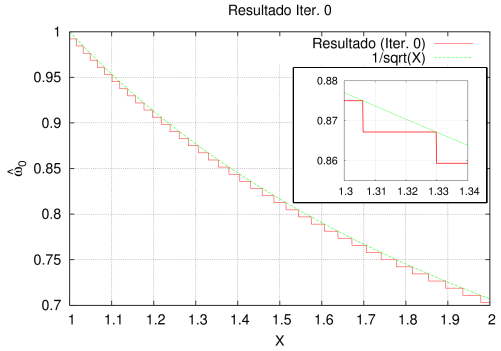
Una vez comprobado el funcionamiento correcto del algoritmo simulado, se realizó la validación del modelo teórico para el error. Se trataba de comprobar que el error obtenido de manera teórica mediante la evaluación de la ecuación 2.28 es el mismo que el medido en la simulación de la propia implementación. Al mismo tiempo, se comprueba la validez de los límites obtenidos para el error de la aproximación al resultado en cada iteración puesto que son derivados del mismo análisis del error.

En la figura 2.12 se pueden ver las dos gráficas. El error obtenido del modelo teórico coincide con el error medido en las simulaciones. En ambas gráficas se puede apreciar también los valores obtenidos para los límites del error en la primera iteración. Del mismo modo que en el recíproco estos han sido obtenidos de sustituir $p = 23$ y $b = 7$ en las ecuaciones 2.55. El valor obtenido para estos límites es correcto y por lo tanto están validados.

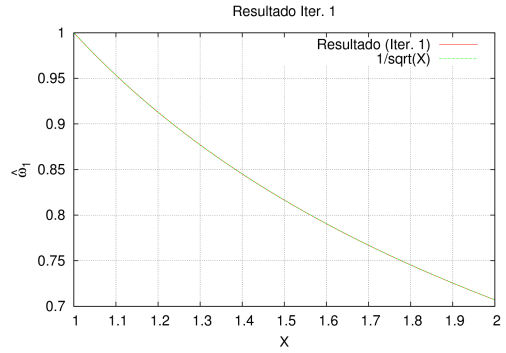
El mismo procedimiento se siguió para la segunda iteración. El modelo teórico vuelve a reproducir de manera adecuada el error de la aproximación al resultado para esta iteración. Igual que sucedía para el recíproco, en la segunda iteración aumenta de manera considerable el número de puntos con error positivo. Esto es debido a un mayor peso de los términos del error de las operaciones intermedias respecto del error de aproximación del algoritmo. Vuelven a mostrar un comportamiento satisfactorio los límites del error obtenidos a partir de la ecuación 2.70.

Al igual que para el caso del recíproco representamos el error en escala logarítmico para poder ver a partir de qué bit fraccional el error afecta al resultado.

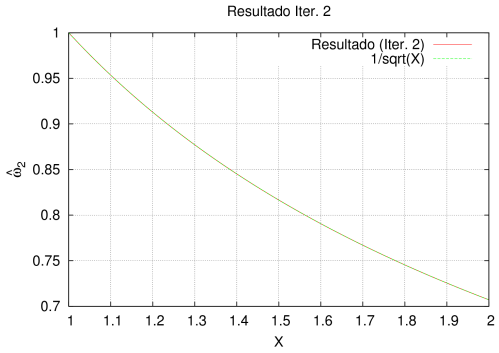
Una vez comprobado que el modelo funcionaba y había sido validado se procedió a aplicar este método descrito al diseño de una unidad bien conocida en la literatura y comparar los resultados obtenidos con los del diseño original.



(a) Resultado de la raíz cuadrada recíproca para la Iteración 0

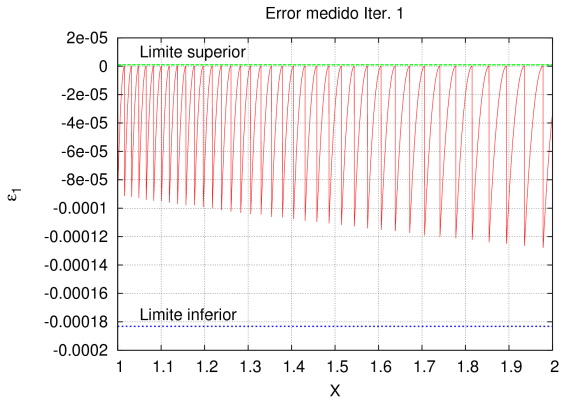


(b) Resultado de la raíz cuadrada recíproca para la Iteración 1

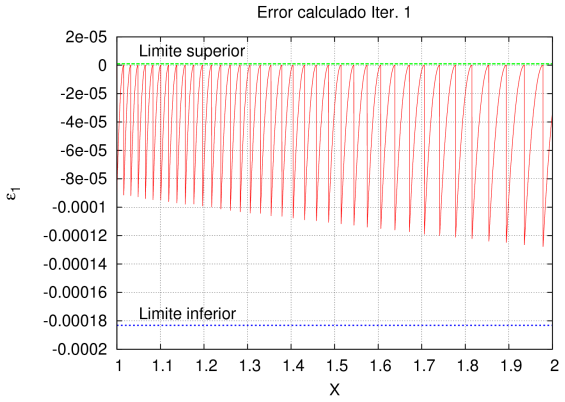


(c) Resultado de la raíz cuadrada recíproca para la Iteración 2

Figura 2.11: Resultado de la raíz cuadrada recíproca en las distintas iteraciones para el algoritmo GLD

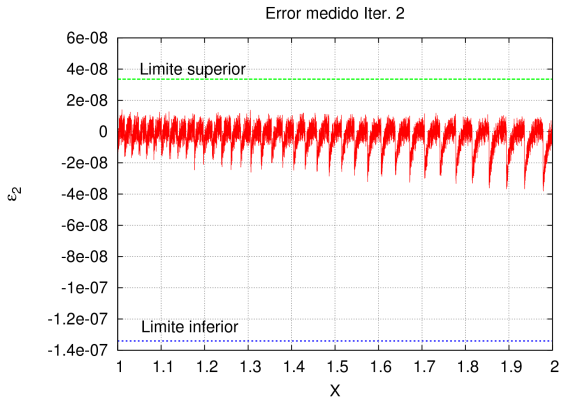


(a) Error medido en la primera iteración

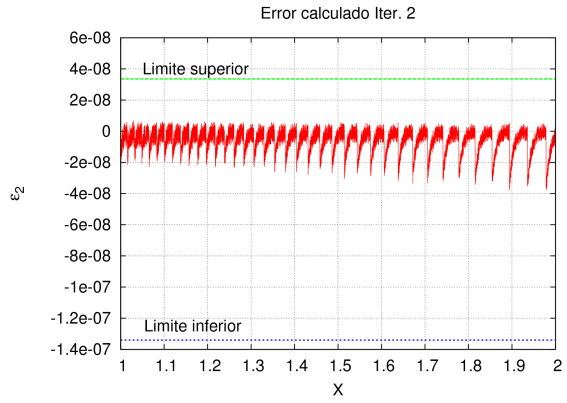


(b) Error en la segunda iteración obtenido de las expresiones teóricas

Figura 2.12: Errores de la aproximación al resultado para la primera iteración de la raíz cuadrada recíproca

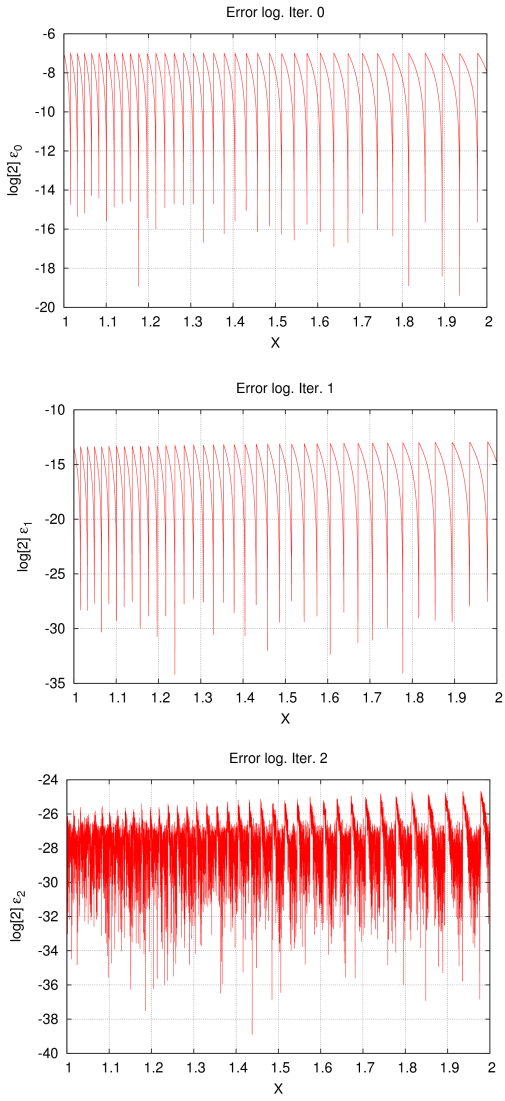


(a) Error medido en la segunda iteración



(b) Error en la segunda iteración obtenido de las expresiones teóricas

Figura 2.13: Errores de la aproximación al resultado para la segunda iteración de la raíz cuadrada recíproca



(a) Error logarítmico para la Iteración 0 de la raíz cuadrada recíproca

(b) Error logarítmico para la Iteración 1 de la raíz cuadrada recíproca

(c) Error logarítmico para la Iteración 2 de la raíz cuadrada recíproca

Figura 2.14: Error logarítmico de la raíz cuadrada recíproca en las distintas iteraciones para el algoritmo de Goldschmidt

División			
	AMD-K7	Seidel [ES03a]	Este trabajo
SP (24 bit)	30x30	–	27x27
DP (24 bit)	59x59	57x60	56x56
Raíz cuadrada			
SP (24 bit)	30x30	–	27x27
DP (24 bit)	59x59	–	56x56

Tabla 2.1: Tamaño del multiplicador para diferentes métodos y formatos aplicados a la unidad de punto flotante del AMD-K7

2.5. Aplicación del método

En esta última sección, este método se aplica al análisis del diseño de la unidad de división y raíz cuadrada para punto flotante en el procesador AMD-K7 [Obe99]. Se ha escogido esta unidad por ser una aplicación paradigmática de los algoritmos multiplicativos aunque podría ser aplicado a cualquier otro. El procesador AMD-K7 utiliza cuatro formatos diferentes para punto flotante: precisión simple (24 bits de mantisa, 8 bits de exponente), precisión doble (53 bits de mantisa, 11 bits de exponente), precisión extendida (64 bits de mantisa, 15 bits de exponente) y un formato de mayor precisión interna (68 bits de mantisa, 18 bits de exponente). El algoritmo usado para calcular la división y la raíz cuadrada es también el algoritmo GLD.

La arquitectura del procesador AMD-K7 utiliza un multiplicador de tamaño 76×76 bits y una aproximación inicial de 15 bits para la división y de 16 bits para la raíz cuadrada. Este tamaño de multiplicador se debe a que está calculado para el formato de precisión más ancho que soporta la unidad, esto es, 68 bits de mantisa. Sin embargo, en este capítulo vamos a realizar las comparaciones de los resultados obtenidos para los formatos de precisión simple y doble. Se puede calcular también el tamaño mínimo para el multiplicador según [Obe99]. Estos datos será los que se comparen con los del método descrito aquí (ver Tabla 2.1).

La unidad de punto flotante del procesador AMD-K7 realiza el cálculo para los formatos de precisión simple y precisión doble en una y dos iteraciones respectivamente. El resto de los formatos de precisión son calculados en la tercera iteración. Nuestro método se ha descrito para dos iteraciones. Por lo tanto solo tiene sentido compararlo con los resultados para los formatos de precisión simple y doble. En la literatura existe otro método para el cálculo del tamaño del multiplicador [ES03a], únicamente para la división en punto flotante para

precisión doble. Este método realiza una análisis del error que obtiene únicamente límites para el error relativo de los distintos parámetros calculado por el algoritmo, es decir, no es exacto. Estos límites son usados para obtener el valor máximo del tamaño de cada multiplicando en las operaciones intermedias. Para minimizar esto se utilizan métodos de redondeo dirigido en dichas operaciones. Los resultados de este trabajo se incluyen también en la comparación.

2.5.1. División

Para el diseño de la unidad de división, utilizaremos los límites obtenidos para el recíproco para una (ecuación (2.35)) y dos iteraciones (ecuación (2.46)). Estos límites necesitan ser transformados para la división. Para obtener el error de la división utilizando el del recíproco no hay más que multiplicarlo por el valor del dividendo (Y), tal y como se explicó anteriormente. En el caso de los límites será necesario multiplicarlos por el valor máximo que el dividendo puede alcanzar, que es 2. De este modo los límites del error para la división serán, para una iteración:

$$-2^{-2b+2} - 2^{-p} \cdot (3 + 2^{-b+1}) \leq \varepsilon_1 \leq 2^{-p} \cdot (3 + 2^{-b+1}) \tag{2.73}$$

Y para dos:

$$-2^{-4b+4} - 2^{-p} \cdot (3 + 7 \cdot 2^{-2b+2}) \leq \varepsilon_2 \leq 2^{-p} \cdot (3 + 2^{-2b+2}) \tag{2.74}$$

Para el diseño de esta unidad en [Obe99] se fija el error máximo del resultado como:

$$-2^{-(n+2)} \leq \omega - \hat{\omega} \leq 2^{-(n+2)} \tag{2.75}$$

donde n es la precisión objetivo. Hay que tener en cuenta que el formato de simple precisión en esta unidad se calcula en una iteración y con una aproximación inicial de 15 bits [Obe99]. Entonces, el límite del error en la primera iteración de acuerdo con (2.75) y con la precisión del formato simple, será el siguiente.

$$|\varepsilon_1| \leq 2^{-25} \tag{2.76}$$

Igualando este valor límite a los límites de la ecuación (2.73) se obtienen dos valores máximos para p. El valor final de p se obtiene escogiendo el más restrictivo de los dos. En este caso es 27 como se ve en la tabla 2.1.

En el caso de la precisión doble se necesitan dos iteraciones y el mismo tamaño de aproximación inicial que en el caso anterior. Luego, ahora se utilizarán las desigualdades de la ecuación (2.74). Usando (2.75) y la precisión final para el formato doble, se obtiene el requerimiento para el error.

$$|\varepsilon_1| \leq 2^{-54} \tag{2.77}$$

Sustituyendo el valor en la expresión de los límites se obtiene un valor óptimo de $p = 56$. De este modo se han calculado los tamaños óptimos para las operaciones intermedias para la división en los formatos de precisión simple y doble definidos por el estándar IEEE754.

2.5.2. Raíz Cuadrada

El método utilizado para el caso de la raíz cuadrada es completamente análogo al utilizado para la división. El primer paso es adaptar los límites obtenidos para la raíz cuadrada recíproca para la raíz cuadrada. El error para la raíz cuadrada se obtiene multiplicándolo por el operando X. Los límites se obtienen multiplicando los anteriores por el valor máximo del operando (2). Los nuevos límites para la primera iteración son:

$$\begin{aligned} \varepsilon_1 &\leq 2^{-3b+2} + \frac{9}{2} \cdot 2^{-p} + \frac{7}{2} \cdot 2^{-p} \cdot 2^{-b} \\ \varepsilon_1 &\geq -3 \cdot 2^{-2b+1} - \frac{9}{2} \cdot 2^{-p} - \frac{7}{2} \cdot 2^{-p} \cdot 2^{-b} \end{aligned} \tag{2.78}$$

Para la segunda:

$$-2^{-4b+5} + \frac{9}{2} \cdot 2^{-p-1} \leq \varepsilon_2 \leq 2^{-6b+6} + \frac{9}{2} \cdot 2^{-p-1} \tag{2.79}$$

Los requerimientos para la exactitud del resultado para este diseño son [Obe99]:

$$-2^{-(n+1)} \leq \omega - \hat{\omega} \leq 2^{-(n+1)} \tag{2.80}$$

La diferencia de estos límites respecto de los de la división los explica el intervalo del resultado de la operación. En el caso de la división se necesitaba un bit más para normalizar el resultado. En este caso, se utiliza una semilla de $b = 16$ bits para la raíz cuadrada. Al igual que para la división, la precisión

simple se obtiene con una iteración y la doble con dos. De acuerdo con esto, el límite del error para el formato de simple precisión será:

$$|\varepsilon_1| \leq 2^{-24} \tag{2.81}$$

Igualando este valor a las expresiones de los límites (2.78) se obtiene un valor para el tamaño de las operaciones intermedias de $p = 27$.

Estableciendo, por el mismo procedimiento, la exactitud para el error en la segunda iteración para obtener el formato de doble precisión.

$$|\varepsilon_1| \leq 2^{-53} \tag{2.82}$$

El valor obtenido para p en este caso es de 56 bits. Todos los tamaños para cada formato están resumidos en la tabla 2.1. Estos tamaños afectan directamente al tamaño del multiplicador, de manera que, utilizando el método propuesto, se producen mejoras significativas en el tamaño y el número de productos parciales. Para el caso de la división se produce una reducción del 10 % en el tamaño y el número de productos parciales respecto del diseño original. Esta reducción es de un 5 % en el caso de doble precisión del tamaño original. Es también posible comparar los datos de doble precisión en división con los del método descrito en [ES03a]. En este caso la reducción del tamaño del tamaño de los productos parciales es del 2 % y el del número de ellos es de un 6 %. En el caso de la raíz cuadrada las reducciones son del 10 % y del 5 % para precisión simple y precisión doble respectivamente, tanto para el tamaño de los productos parciales como para su número. Estas mejoras son debidas a que el método se construye a partir de un análisis del error más exacto. La mejora en estas características del multiplicador producirá reducciones significativas en el hardware, el retardo y la potencia necesaria en el diseño hardware de los algoritmos.

2.6. Conclusión

En este capítulo se ha presentado un método de diseño para unidades de división y raíz cuadrada basadas en el algoritmo GLD. Este método permite obtener el tamaño de palabra óptimo para las operaciones intermedias. Este tamaño determina de manera exacta el tamaño del multiplicador. Dicho tamaño se ofrece en base a la longitud de los productos parciales y al número de ellos.

Este método está basado en un análisis preciso del error que tiene en cuenta las diferentes contribuciones al error. Estas contribuciones consisten en el error de aproximación del algoritmo, por un lado, y la contribución al error de las

operaciones intermedias. Esta última es provocada por el error de redondeo interno de las unidades que realizan dichas operaciones. Este análisis obtiene una expresión analítica del error final de una iteración cualquiera. A través de esta expresión se han calculado los límites para los valores del error en cada iteración.

Para probar la validez del análisis del error se ha simulado la ejecución en hardware del algoritmo. Esta simulación se ha realizado para un esquema de dos iteraciones y el formato estándar de precisión simple. Al hacer las simulaciones, se ha comprobado que el error que reproducen las expresiones analíticas y el error medido son exactamente iguales De este modo queda validado el análisis del error. Al mismo tiempo fueron validados también los límites obtenidos para el error. Estos reproducen bien el comportamiento del error pudiendo ser utilizados para el diseño de la implementación hardware del algoritmo. Utilizando los requerimientos del error para cada diseño en cuestión, se puede obtener la longitud del resultado para las operaciones intermedias.

Finalmente, el método descrito en este capítulo es aplicado a la unidad de división y raíz cuadrada en punto flotante del procesador AMD-K7. Se muestran los resultados de este nuevo método comparados con el propio diseño original del AMD-K7 y, en el caso de la división en precisión doble, con un método adicional. En esta comparación se demuestra que existen reducciones significativas en el tamaño del multiplicador. Este método puede ser aplicado a otros algoritmos multiplicativos. La descripción del método para el algoritmo NR se presenta en el Apéndice A.

Capítulo 3

Mejora del método tradicional de redondeo para latencia variable

En este capítulo se presenta un nuevo método de redondeo de latencia variable que es más eficiente que el método tradicional de redondeo de algoritmos multiplicativos. El método tradicional de redondeo consiste en la suma de una constante al resultado obtenido del algoritmo. El resultado de la suma se trunca para obtener una aproximación al resultado con bits extra respecto de la precisión exigida. A continuación se calcula el resto. Examinando el bit extra y el valor del resto se decide la acción de redondeo. El método propuesto aquí está basado en el método tradicional. Se incluyen bits adicionales en la decisión de la acción de redondeo. Estos bits pertenecen al resultado obtenido del algoritmo antes de realizar la suma de la constante. Utilizando estos bits se reduce el número de casos en los que el cálculo del resto es necesario. Este método ha sido validado por medio de simulaciones exhaustivas. Dichas simulaciones permiten medir exactamente cuál es el número de veces en que se evitan el cálculo del resto. La conclusión que se extrae es que este método es capaz de reducir a la mitad el numero de veces en que se necesita el cálculo del resto. Utilizando una aproximación al resultado adecuado, este cálculo se realiza solamente en el 5 % del número total de casos.

3.1. Introducción

Se propone aquí una modificación nueva de los algoritmos de redondeo de latencia variable explicados en las secciones 1.3.1 y 1.3.2. La alternativa típica para el redondeo de algoritmos de convergencia cuadrática consiste en crear un aproximación al resultado. Esta aproximación se usa para calcular el resto. De la comparación del valor del resto con cero y de los últimos bits de la aproximación se extrae la decisión para realizar la acción de redondeo. Esta es la solución para el redondeo más comúnmente usada.

Los método de redondeo propuestos en las secciones 1.3.1 y 1.3.2 son dos mejoras al método inicialmente propuesto. Ambos tratan de reducir el número de casos en los que se necesita el cálculo del resto para realizar el redondeo. Para estos casos en los que no se calcula el resto la latencia total del algoritmo se reduce de manera significativa. Estos métodos de redondeo son, evidentemente, de latencia variable. Este tipo de algoritmos tienen sentido, especialmente, el en contexto de procesadores con planificación dinámica.

Sin pérdida de generalidad, se explicará en este capítulo el método para el redondeo al más próximo por simplicidad. Las conclusiones extraídas para este modo de redondeo serán, también, válidas para el resto de los modos de redondeos definidos en el estándar IEEE 754.

Al igual que en las secciones anteriores, se asumirá que el resultado con precisión infinita se denotará por ω. El objetivo del redondeo es obtener una aproximación al resultado redondeado correctamente con n bits de precisión. La aproximación obtenida del algoritmo se denota como $\hat{\omega}$. Esta aproximación tiene una precisión mayor. Se asume también que $\hat{\omega}$ tiene un error que puede ser positivo o negativo como es usual en las implementaciones hardware de los algoritmos multiplicativos.

Con este método se consigue una reducción dramática de los casos donde el cálculo del resto es necesario. Con un número suficiente de bits extra en la aproximación obtenida del algoritmo, se necesita calcular el resto en un porcentaje muy bajo de los casos. Este método ha sido validado con simulaciones exhaustivas.

Esta modificación se basa en la búsqueda de información adicional que permita discriminar más casos con una acción de redondeo fija, es decir, independiente del valor del resto. Se obtiene esta información de $\hat{\omega}$ para intentar reducir el número de casos que son más complejos de redondear. Obtener esta información adicional implica que la aproximación al resultado obtenida ha de ser proporcionada con una exactitud mayor. Consecuentemente, también se necesitará una mayor precisión. Como se verá más tarde, para precisión simple la

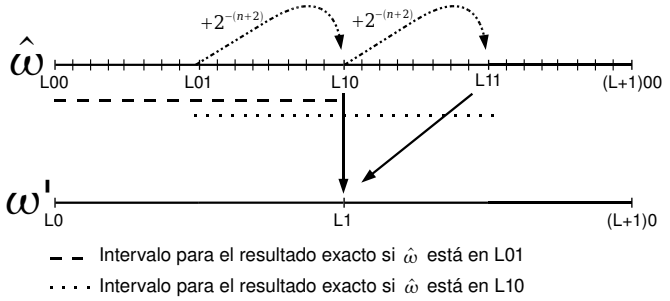


Figura 3.1: Los dos primeros pasos en el redondeo para $j = 2$ y $k = 5$

implementación de este método es posible con $n + 5$ bits de precisión, siendo n la precisión final requerida. Esto supone, incrementar muy ligeramente los requerimientos hardware del diseño.

3.2. Modificación del método clásico de redondeo

En esta sección se aplica el nuevo método al método clásico de redondeo descrito en el apartado 1.3.1. Este método es una modificación del método tradicional de latencia variable para el redondeo de algoritmos multiplicativos [Sch95]. Partiremos, por tanto de las mismas condiciones que dicho método.

Se supone que $\hat{\omega}$ se calcula con $n + k$ bits de precisión y que tiene una exactitud de $\pm 2^{-(n+j)}$, donde $k \geq j \geq 2$.

$$-2^{-(n+j)} < \omega - \hat{\omega} < 2^{-(n+j)} \tag{3.1}$$

El método calcula la aproximación al resultado ω' con $n+1$ bits de precisión y una exactitud de $1ulp$. El redondeo se realiza examinando el bit $n+1$ de ω' y el resto cuando es necesario. En la figura 3.1 está representada una aproximación al resultado calculada con aproximación al resultado $\hat{\omega}$ se ha sido obtenida con $n + 2$ bits de exactitud.

$$-2^{-(n+2)} < \omega - \hat{\omega} < 2^{-(n+2)} \tag{3.2}$$

Es decir, siguiendo la notación utilizada anteriormente $j = 2$. Además, $k = 5$, de manera que el resultado del algoritmo ha sido calculado con una precisión de $n+5$ bits. Esta es la precisión necesaria para obtener este mismo resultado con la exactitud mencionada antes. La figura 3.1 representa los dos primeros pasos del proceso de redondeo para la aproximación al resultado obtenida de un algoritmo multiplicativo calculado con un bits extra de exactitud. Se representan todos los valores que puede tomar $\hat{\omega}$ entre dos puntos de precisión n que se diferencian en 1 ulp. Los puntos etiquetados se representan con dos bits por simplicidad de la figura. L es el bit en la posición n, luego habría que añadirle dos bits más. Por ejemplo, la posición $L10$ es el número $1.xx....xL10000$. Hay 8 posibles valores entre $L10$ y $L11$ que son de la forma $1.xx....xL10xxx$, desde $1.xx....xL10000$ a $1.xx....xL10111$.

Sólo están representados los pasos de redondeo para para los puntos $\hat{\omega}[n : n+5] = L01000$ y $\hat{\omega}[n : n+5] = L10000$. La razón de esto es que, si se observa la tabla 1.2, es fácil darse cuenta de que estos son precisamente los casos para los cuales el resto ha de ser calculado.Después de incrementar y truncar a $n+1$ bits cada uno de ellos, ambos se convierten en el mismo punto $\omega'[n : n+1] = L1$ en la línea que representa a ω'. Lo mismo ocurre para todos los puntos con $\hat{\omega}[n : n+2] = L01$ y $\hat{\omega}[n : n+2] = L10$. Por esta razón, nos centraremos a partir de ahora, en estos puntos.

Método clásico: A continuación enumeramos los pasos en que consiste el método clásico:

- **Paso 1:** Incrementar $\hat{\omega}$ en una constante: $2^{-(n+1)}$
- **Paso 2:** Truncar el resultado obtenido a $n+1$ bits.
- **Paso 3:** Determinar si el cálculo del resto es necesario. Esto se hace mediante la inspección de la tabla de redondeo (Tabla 1.7).
- **Paso 4:** Cálculo del resto (si es necesario) $rem = 1 - X \cdot \omega'$. Observando el signo y la magnitud del resto y el bit $n+1$ de ω' se realiza la acción de redondeo correspondiente de acuerdo con la tabla 1.7.

Método clásico modificado: Se presenta a continuación el nuevo método propuesto. Este método consiste en añadir información que permita reducir el número de casos en los que se realiza el cálculo del resto. Esta información consistirá, como se verá a continuación, en algunos bits de $\hat{\omega}$. Esto supone una modificación del tercer paso del algoritmo clásico para construir una nueva tabla de redondeo que incluya la nueva información adicional. A continuación se presentan los argumentos para poder construir la nueva tabla de redondeo.

Como se mencionó anteriormente, sólo los valores de $\hat{\omega}$ con $\hat{\omega}[n:n+2] = L01$ y $\hat{\omega}[n:n+2] = L10$ se convierten en el punto $\omega'[n:n+1] = L1$. Cuando $\hat{\omega}[n:n+5] = L01000$ el resultado exacto puede estar en el intervalo representado en la figura 3.1 como una línea discontinua. La ecuación (3.2) puede también ser aplicado al resto de los puntos con $\hat{\omega}[n:n+2] = L01$. Para estos puntos el resultado exacto puede ser mayor o menor que $\omega'[n:n+1] = L1$. Usando las simulaciones exhaustivas en la sección siguiente, se determinó que para algunos de estos puntos el resultado es siempre menor que $\omega'[n:n+1] = L1$. Estos puntos son $\hat{\omega}[n:n+5] = L01000$, $L01001$, $L01010$, $L01011$. Para estos puntos el resultado exacto es siempre menor que el valor en $\omega'[n:n+1] = L1$ y el resto tiene siempre el mismo signo. Consecuentemente, para estos casos particulares, la acción de redondeo es siempre la misma.

La otra posibilidad son los puntos con $\hat{\omega}[n:n+2] = L10$. Ahora, la línea punteada en la figura 3.1 representa el intervalo donde puede estar el resultado exacto para $\hat{\omega}[n:n+2] = L10000$. El valor en $\omega'[n:n+1] = L1$ puede ser mayor o menor que el resultado exacto y, por lo tanto, el resto puede ser positivo y negativo. Para este caso, el valor del resto sí que importa para determinar la acción de redondeo. Estos dos posibles casos pueden ser distinguidos usando los bits $n+1$ y $n+3$ de $\hat{\omega}$.

Teniendo en cuenta estas conclusiones, es posible construir una nueva tabla de redondeo (Tabla 3.1). Esta tabla introduce dos columnas adicionales respecto a la clásica (Tabla 1.2), los bits $n+1$ y $n+3$ de $\hat{\omega}$.

A continuación se describen los pasos del algoritmo de redondeo modificado. Son los siguientes:

- **Paso 1:** Sumar $2^{-(n+2)}$ a $\hat{\omega}$.
- **Paso 2:** Truncar el resultado a $n+1$ bits. Esto permite el cálculo de la aproximación ω' con un bit extra de exactitud. Estos dos pasos son los mismos que en el método tradicional.
- **Paso 3:** Decidir si el resto debe ser calculado o no. Esto se hace mediante la inspección de la tabla de redondeo (Tabla 3.1).

ω'[n+1]	$\hat{\omega}$[n+1]	$\hat{\omega}$[n+3]	rem	acción de redondeo
0	–	–	–	truncar
1	0	–	=0	–
1	0	0	–	truncar
1	0	1	–	**truncar**
1	0	1	+	**incrementar**
1	1	–	-	**truncar**
1	1	–	+	**incrementar**

Tabla 3.1: Nueva tabla de redondeo con un bit extra.

- **Paso 4:** Calcular el resto cuando es necesario. Sólo dos casos en la tabla 3.1. En función del valor de resto y de los bits correspondientes realizar la acción de redondeo.

Como se puede ver el algoritmo es análogo al tradicional. Solamente cambia la tabla de redondeo que incluye algunos bits adicionales para tomar la decisión de determinar la acción de redondeo.

3.3. Modificación del método clásico de redondeo con más de un bit extra

La misma metodología puede ser aplicada a un método que usa una aproximación con dos bits extra de exactitud (Sección 1.3.2). Este método es una extensión del anterior que trata de incrementar el número de casos donde el cálculo del resto no es necesario. Esto se hace a través del cálculo de ω' con más de un bit extra. En esta sección se propone una modificación, similar a la de la sección anterior, para esta extensión del método clásico (Figura 3.2).

En este método la exactitud de $\hat{\omega}$ es

$$-2^{-(n+3)} < \omega - \hat{\omega} < 2^{-(n+3)} \tag{3.3}$$

Esta es la precisión mínima necesaria para obtener una aproximación al resultado con dos bits extra de exactitud respecto de la requerida.

Método clásico: Los pasos del método clásico eran los siguientes:

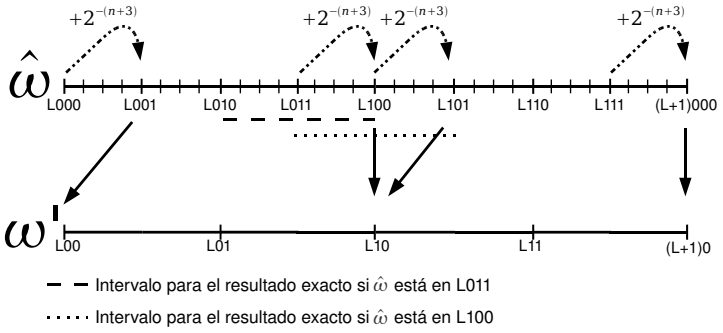


Figura 3.2: Los dos primeros pasos en el redondeo para

- **Paso 1:** Sumar a $\hat{\omega}$ la constante $2^{-(n+3)}$.
- **Paso 2:** Truncar el resultado a $n+2$ para formar ω'. La exactitud de ω' es ahora

$$-2^{-(n+2)} < \omega - \omega' < 2^{-(n+2)} \tag{3.4}$$

  De este modo, se calcula la aproximación ω' con dos bits extra.
- **Paso 3:** Decidir si el resto debe ser calculado o no. Esto se hace mediante la inspección de la tabla de redondeo (Tabla 1.8).
- **Paso 4:** Calcular el resto $rem = 1 - X \cdot \omega'$ (si es necesario). Examinando el signo y la magnitud del resto y los dos bits extra de ω' se pueden implementar el método de redondeo de acuerdo con la tabla 1.8.

Método clásico modificado: El método es similar al propuesto para una aproximación con un bit extra. Ahora tenemos dos bits extra de exactitud. También en este caso se busca información adicional en la aproximación $\hat{\omega}$ para reducir el número de veces en que se calcula el resto. La explicación de como se puede conseguir esto se puede extraer de la figura 3.2.

La figura 3.1 representa los dos primeros pasos del proceso de redondeo para la aproximación al resultado obtenida de un algoritmo multiplicativo calculado

ω'[n+1:n+2]	ω'[n+1]	$\hat{\omega}$[n+4]	rem	acción de redondeo
00	–	–	–	truncar
01	–	–	–	truncar
10	–	–	=0	–
10	0	0	–	truncar
10	0	1	-	**truncar**
10	0	1	+	**incrementar**
10	1	–	-	**truncar**
10	1	–	+	**incrementar**
11	–	–	–	incrementar

Tabla 3.2: Nueva tabla de redondeo con dos bits extra.

con dos bits extra de exactitud. Se representan todos los valores que puede tomar $\hat{\omega}$ entre dos puntos de precisión n que se diferencian en 1 ulp. Los puntos etiquetados se representan con tres bits por simplicidad de la figura. L es el bit en la posición n, luego habría que añadirle dos bits más. Por ejemplo, la posición $L100$ es el número $1.xx....xL10000$. Hay 5 posibles valores entre $L100$ y $L101$ que son de la forma $1.xx....xL10xxx$, desde $1.xx....xL10000$ a $1.xx....xL10011$.

Una vez más, como en el caso anterior, solamente los valores de $\hat{\omega}$ con $\hat{\omega}[n : n+3] = L011$ y $\hat{\omega}[n : n+3] = L100$ producen el mismo valor $\omega'[n : n+2] = L10$. Igualmente, cuando $\hat{\omega}[n : n+3] = L011$ el resultado exacto es siempre menor que $\omega'[n : n+2] = L10$ para algunos valores de $\hat{\omega}$. Para $\hat{\omega}[n : n+3] = L01100$, $L01101$ el signo del resto de $\omega'[n : n+2] = L10$ es siempre el mismo y, por lo tanto, la acción de redondeo es también siempre la misma. De manera opuesta, cuando $\hat{\omega}[n : n+3] = L100$ el resultado exacto puede ser mayor o menor que el valor de ω' en $\omega'[n : n+2] = L10$ para todos los casos. Este caso requiere el cálculo del resto puesto que su signo puede cambiar. Dicho cambio de signo fuerza una acción de redondeo diferente. Podemos detectar estos casos fijándonos en los bits $n+1$ y $n+4$ de $\hat{\omega}$. El problema es muy similar al de la sección anterior y también se puede crear una nueva tabla de redondeo para este caso (Tabla 3.2). El número de veces en que se calcula el resto se reduce a la mitad respecto del caso anterior.

- Sumar $2^{-(n+3)}$ a $\hat{\omega}$.
- En este caso, se trunca a $n+2$ bits para permitir el cálculo de la aproximación ω' con dos bits extra de exactitud.

- Determinar cuando el cálculo del resto es necesario para el redondeo mediante la tabla 3.2.
- Cálculo del resto (si es necesario) y acción de redondeo. La acción de redondeo se determina escogiendo la entrada de la tabla 3.2 de acuerdo con el valor del resto y de los dos bits extra de ω'.

Al igual que en la sección 1.3.2 este método puede ser extendido y aplicado a mayor número de bits extra en la aproximación al resultado.

3.4. Validación del método

Los métodos de redondeo mencionados en las secciones 1.3.1 y 1.3.2, propuestos respectivamente en [Sch95] y [OF97], no determinan de manera cuantitativa el número de casos exacto donde el cálculo del resto es necesario y dónde no lo es. Se realiza una medida cualitativa de la reducción de casos observando la fracción de posibles entradas en la tabla de redondeo que requieran la multiplicación y la resta necesarias para el cálculo del resto. Pero esta fracción no coincide con el porcentaje exacto de casos, respecto del número total de casos posibles medidos en una implementación.

Se usaron simulaciones exhaustivas para tener una medida exacta en la reducción del número de operaciones y para determinar exactamente qué puntos necesitan del cálculo del resto para ser redondeados. Todas las simulaciones fueron realizadas con Maple [GLM08]. Estas simulaciones reproducen la ejecución en hardware de los algoritmos multiplicativos para configuraciones diferentes y la ejecución de la función de redondeo. De entre los algoritmos multiplicativos se escogió el algoritmo de Goldschmidt, aunque se pueden obtener los mismos resultados con el algoritmo de Newton-Raphson, puesto que el método descrito es independiente del algoritmo utilizado. La operación aritmética simulada fue el recíproco para precisión simple [1]. Las simulaciones para otras operaciones proporcionan resultados muy similares. Todas las simulaciones son exhaustivas, es decir, se simulan todos los posibles operandos de la implementación. Para determinar si el tipo de implementación tiene alguna influencia en los resultados se realizaron simulaciones para los dos esquemas diferentes de la figura 1.5.

El número de casos donde el cálculo del resto es necesario fue medido para los métodos de las secciones 1.3.1 (Método 1) y 1.3.2 (Método 2). Estos datos se

[1] Las simulaciones para precisión doble consumen un tiempo y unos recursos computacionales inasumibles. La precisión simple permite la simulación en un tiempo razonable y proporciona datos igualmente significativos.

	Método 1 (%)	Método 1 modificado (%)	Method 2 (%)	Método 2 modificado (%)
1 iteración	25.01	15.51	12.49	9.49
2 iteraciones	25.00	15.50	12.48	9.24

Tabla 3.3: Número de veces en los que se calcula el resto para recíproco y raíz cuadrada

comparan con el número de veces en que el cálculo del resto es necesario usando el método nuevo explicado en esta sección (Método modificado).

Los resultados obtenidos de estas simulaciones están resumidos en la tabla 3.3. En esta tabla se puede ver que el número de iteraciones del algoritmo no afecta a la cantidad de veces en que el cálculo del resto es necesario. Los porcentajes son prácticamente iguales para las filas de 1 y 2 iteraciones. Sin embargo, el número de bits extra utilizados en la aproximación al resultado si que produce reducciones significativas en los porcentajes, tal y como se esperaba. El Método 1 necesita el cálculo del resto en el 25 % de los casos mientras que el Método Modificado reduce este porcentaje hasta un 15 % (para una y dos iteraciones). El Método 2 mejora los resultados del Método 1 utilizando dos bits adicionales en la aproximación al resultado. De este modo sólo calcula el resto en el 12 % de los casos. Introduciendo la modificación propuesta los resultados también mejoran (9 % de todos los casos). Es importante destacar que esta modificación reduce el número de veces en que se calcula el resto en todos los casos. Otra consecuencia importante que se extrae es que la modificación de métodos que utilizan más bits en la aproximación producirá reducciones aún mayores. Utilizando las predicciones del método descrito en la sección 1.3.2 y aplicando el método propuesto en esta sección se pueden llegar hasta porcentajes de un 5 %. Esto lleva al algoritmo a reducir drásticamente su latencia la gran mayoría de los casos.

3.5. Conclusión

Se ha presentado un método de redondeo que reduce la latencia de los algoritmos multiplicativos en la mayoría de los casos. Este método permite una reducción de los casos donde el cálculo del resto es necesario en un 40 % de los casos. De este modo, el cálculo del resto sólo ocurre en un 15 % de los casos cuando se usa un bit extra en la aproximación al resultado. Este porcentaje se reduce al 9 % cuando se usan dos bits en dicha aproximación. Si se utilizasen

3 bits adicionales este porcentaje podría reducirse hasta un 5 %. En todos los casos esta modificación siempre produce un algoritmo de latencia variable.

El resultado obtenido del algoritmo se modifica para obtener otra aproximación con algunos bits extra además de aquellos requeridos por la precisión mínima necesaria. El cálculo del resto se evita en algunos casos. El método propuesto aquí incluye en la decisión de la acción de redondeo bits del resultado obtenido directamente del algoritmo. Examinando estos bits, además de los ya tradicionalmente usados, permiten una reducción drástica en el número de veces en que es necesario el cálculo del resto.

Se han llevado a cabo simulaciones exhaustivas de diferentes implementaciones de un algoritmo multiplicativo. Estas simulaciones permiten la medición rigurosa del número de veces en que se calcula el resto. Los resultados obtenidos de ellas muestran importantes reducciones en dicho número.

Capítulo 4

Redondeo con cálculo en paralelo del resto

Según el estándar IEEE 754 los resultados de los algoritmos multiplicativos para obtener la división, la raíz cuadrada y sus recíprocos deben de estar redondeados. Este redondeo requiere el cálculo del resto. Este cálculo produce una penalización muy fuerte en el tiempo de ejecución de este tipo de algoritmos. En este capítulo se presenta un método de redondeo de latencia variable basado en el cálculo del resto en paralelo con la ejecución del algoritmo (para los algoritmos de Newton-Raphson (NR) y Goldschmidt (GLD)). Este método produce el resto del resultado que se obtiene directamente del algoritmo $\hat{\omega}$, sin ninguna transformación. Debido a este hecho se presenta un nuevo método de redondeo que trabaja directamente con esta aproximación al resultado. Este método, debido al error introducido por las operaciones intermedias, evita el cálculo del resto de manera tradicional y su penalización en tiempo en la gran mayoría de los casos, pero no en todos. Para el caso de la división únicamente será necesario el cálculo del resto de manera tradicional en el 9 % de los casos para la división y el 18 % en el caso de la raíz cuadrada recíproca. Estos resultados mejoran los obtenidos para división en otras trabajos previos. En el caso de la raíz cuadrada y la raíz cuadrada recíproca mejora algunos métodos y otros no.

4.1. Introducción

La exigencia del estándar IEEE 754 obliga a que los resultados del recíproco, la división, la raíz cuadrada y la raíz cuadrada recíproca sean sometidos a un redondeo. Se debe proporcionar la posibilidad de que dicho resultado pueda ser redondeado según uno de los cuatro modos de redondeo existentes: redondeo hacia $+\infty$, redondeo hacia $-\infty$, redondeo hacia cero o redondeo al más próximo. Para el caso de los algoritmos sustractivos este hecho no supone ningún problema, puesto que las propias iteraciones de los algoritmos producen el resto en cada una de sus iteraciones. Desafortunadamente, las iteraciones de los algoritmos multiplicativos producen una aproximación al resultado pero no producen el resto. Esto hace que el redondeo en los algoritmos multiplicativos sea un problema y sea uno de los principales aspectos de investigación en este campo.

En los algoritmos multiplicativos, el resto tiene que ser calculado una vez que se ha obtenido la aproximación la resultado. Este cálculo añade a la latencia total el tiempo de una multiplicación para el casos de la división y el recíproco. En el caso de la raíz cuadrada y la raíz cuadrada recíproca el tiempo total penalizado corresponde a dos multiplicaciones.

Existe una solución alternativa que evita el cálculo del resto en los algoritmos multiplicativos [Mar90]. Su principal desventaja es que la aplicación de dicho método implica la obtención del resultado con el doble de exactitud de la requerida. Esto provoca un aumento muy significativo del área empleada en la implementación de los diseños. Además, supone también una iteración adicional del algoritmo para obtener el resultado con la exactitud adecuada. Por todas estas razones, esta alternativa no es la más adecuada.

Existen trabajos previos que intentan reducir la penalización del cálculo del resto. Estos son métodos de latencia variable que evitan el cálculo del resto en algunos casos. En [Sch95] se introdujo por primera vez un método que evita el cálculo del resto para algunos casos. En [OF97] se propone una mejora del método anterior. Se demuestra que, si se usan m bits de guarda en la estimación del cociente, sólo es necesario calcular el resto para una fracción más pequeña de los casos (2^{-m}).

Lo que aquí se propone es un nuevo método para obtener una algoritmo de redondeo de latencia variable. El resto se calcula en paralelo con la ejecución del algoritmo. De este modo, esta manera de calcular el resto no provoca ninguna penalización al tiempo total de ejecución del algoritmo. El resto obtenido de esta forma es el resto del resultado directamente del algoritmo ($\hat{\omega}$). Esto es una diferencia con los métodos mencionados anteriormente en la sección 1.3, que transforman este resultado para obtener una aproximación (ω'). A esta

aproximación transformada es a la que aplican el redondeo. Este hecho provoca que haya que diseñar un nuevo método de redondo basado en $\hat{\omega}$, en lugar de ω'.

Con este esquema se podría esperar la eliminación del cálculo convencional del resto completamente. Sin embargo, el problema es sensiblemente más complicado. El error de las operaciones intermedias hace que el nuevo resto que se calcula no sea exactamente igual al resto convencional. Este nuevo resto produce el resultado redondeado correcto en la mayoría de los casos, pero no en todos.

La validación de este método se realizó por medio de simulaciones exhaustivas. Los resultados de estas simulaciones permiten proponer un nuevo método de redondeo de latencia variable que ofrece mejores resultado que otros métodos propuestos previamente. Dichos resultados muestran que con este método sólo será necesario el cálculo convencional del resto aproximadamente en el 9 % de los casos para el recíproco y la división. Para el caso de la raíz cuadrada y la raíz cuadrada recíproca este porcentaje es el 18 %.

4.2. Cálculo del resto en paralelo

Esta sección esta dedicada a proponer una manera diferente de calcular el resto para los algoritmos GLD y NR. El resto se obtiene en paralelo con la ejecución del algoritmo. Después de explorar diferentes posibilidades se obtuvieron expresiones para el resto en función del resto de la iteración anterior. En secciones posteriores se abordará la validación de estas expresiones.

4.2.1. Resto en paralelo para el algoritmo GLD

Aunque se han explorado otras alternativas, la mejor manera para obtener el resto en paralelo con la ejecución del algoritmo GLD se basa en la expresión convencional del cálculo del resto para una iteración genérica. Comenzamos por el algoritmo para el recíproco explicado en la sección 1.2.2.

$$\begin{aligned} K_i &= 2 - r_{i-1} \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i \end{aligned} \tag{4.1}$$

Recíproco: La expresión del resto para el recíproco viene dada por:

$$rem_i = 1 - X \cdot \omega_i \tag{4.2}$$

Es posible transformar esta ecuación si tenemos en cuenta la ecuación (1.46). Introduciendo esta en la expresión anterior obtenemos:

$$rem_i = 1 - X \cdot \omega_{i-1} \cdot K_i \tag{4.3}$$

A continuación, es necesario obtener el producto $X \cdot \omega_{i-1}$ en función del resto de la iteración anterior. De (4.2):

$$X \cdot \omega_{i-1} = 1 - rem_{i-1} \tag{4.4}$$

Finalmente, se obtiene una expresión del resto en la iteración i en términos del resto de la iteración anterior y de K_i.

$$rem_i = 1 + K_i \cdot (rem_{i-1} - 1) \tag{4.5}$$

De manera teórica, esta expresión permite el cálculo del resto sin tener que esperar hasta que el resultado de la iteración esté disponible. De este modo se puede obtener el valor del resto en paralelo.

La misma expresión para el caso de la división es muy fácil de obtener. Hay que tener en cuenta la ecuación para obtener el resultado de la división (ecuación 4.1) y la expresión del resto para la división, que es la siguiente:

$$rem = Y - X \cdot \omega_i \tag{4.6}$$

Combinando las dos ecuaciones mencionadas anteriormente, se obtiene la nueva ecuación para el resto. Esta consisten en multiplicar la ecuación obtenida para el recíproco por el dividendo Y.

$$rem_i = Y \cdot [1 + K_i \cdot (rem_{i-1} - 1)] \tag{4.7}$$

Raíz cuadrada recíproca: El proceso para la obtención de una expresión alternativa para el resto en el caso de la raíz cuadrada recíproca es análogo a lo explicado anteriormente. El algoritmo para esta operación consiste en:

$$\begin{aligned} K_i &= \frac{1}{2}(3 - r_{i-1}) \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i^2 \end{aligned} \tag{4.8}$$

La nueva expresión está basada en la expresión convencional:

$$rem_i = 1 - X \cdot \omega_i^2 \tag{4.9}$$

Combinando la ecuación anterior con la expresión de ω_i en la ecuación (4.8) se obtiene:

$$rem_i = 1 - X \cdot \omega_{i-1}^2 \cdot K_i^2 \tag{4.10}$$

Al igual que en el caso anterior, se puede relacionar parte de esta expresión con el resto de la iteración anterior.

$$X \cdot \omega_{i-1}^2 = 1 - rem_{i-1} \tag{4.11}$$

Sustituyendo la ecuación anterior en (4.10) se obtiene la expresión que permite obtener el resto en paralelo para la raíz cuadrada recíproca.

$$rem_i = 1 + K_i^2 \cdot (rem_{i-1} - 1) \tag{4.12}$$

En el caso del resto de la raíz cuadrada, se parte de la expresión de su resto.

$$rem_i = X - X \cdot \omega_i \tag{4.13}$$

Siguiendo los mismos pasos que en los casos anteriores se llega a la conclusión de que el nuevo resto se obtiene sin más que multiplicar la ecuación del resto para la raíz cuadrada recíproca por el operando X.

$$rem_i = X \cdot [1 + K_i^2 \cdot (rem_{i-1} - 1)] \tag{4.14}$$

4.2.2. Resto en paralelo para el algoritmo NR

El proceso para obtener las expresiones para el cálculo paralelo del resto es completamente análogo al que se sigue para el algoritmo GLD.

Recíproco: Las ecuaciones correspondientes al algoritmo para el cálculo del recíproco consisten en:

$$\begin{aligned} n_{i-1} &= X \cdot \omega_{i-1} \\ d_{i-1} &= 2 - n_{i-1} \\ \omega_i &= \omega_{i-1} \cdot d_{i-1} \end{aligned} \tag{4.15}$$

El punto de partida es, igualmente, la ecuación del resto del recíproco.

$$rem_i = 1 - X \cdot \omega_i \tag{4.16}$$

Combinando la ecuación anterior con la ecuación (4.15) se obtiene:

$$rem_i = 1 - X \cdot \omega_{i-1} \cdot d_{i-1} \tag{4.17}$$

Una parte de la ecuación anterior se puede igualar a la expresión del resto para la iteración previa:

$$X \cdot \omega_{i-1} = 1 - rem_{i-1} \tag{4.18}$$

Introduciendo esta ecuación en (4.17) se obtiene la expresión final del resto paralelo para el recíproco.

$$rem_i = 1 + d_{i-1} \cdot (rem_{i-1}); \tag{4.19}$$

Si se quiere obtener el resto para la división, se multiplica el resto obtenido anteriormente por el dividendo Y. Esto tiene que ser necesariamente así ya que, como ya se explicó, la única manera de obtener al división para el algoritmo NR es obtener el recíproco y multiplicarlo por el dividendo.

$$rem_i = Y * [1 + d_{i-1} \cdot (rem_{i-1})] \tag{4.20}$$

Raíz cuadrada recíproca: Las ecuaciones de algoritmo para la raíz cuadrada recíproca son las siguientes:

$$\begin{aligned} s_{i-1} &= \omega_{i-1}^2 \\ n_{i-1} &= X \cdot n_{i-1} \\ d_{i-1} &= 3 - n_{i-1} \\ \omega_i &= \frac{\omega_{i-1}}{2} \cdot d_{i-1} \end{aligned} \tag{4.21}$$

En el caso de la raíz cuadrada recíproca, partimos del resto:

$$rem_i = 1 - X \cdot \omega_i^2 \tag{4.22}$$

La combinación de la ecuación anterior y de la ecuación (4.21) permite obtener la expresión

$$rem_i = 1 - X \cdot \omega_{i-1}^2 \cdot d_{i-1}^2 \tag{4.23}$$

El producto $X \cdot \omega_{i-1}$ puede ser obtenido en función del resto de la iteración anterior.

$$X \cdot \omega_{i-1} = 1 - rem_{i-1} \tag{4.24}$$

Combinando estas dos últimas ecuaciones obtenemos la expresión buscada para el resto del recíproco de la raíz cuadrada:

$$rem_i = 1 + d_{i-1}^2 \cdot (rem_{i-1} - 1) \tag{4.25}$$

Para obtener el resto de la raíz cuadrada únicamente hay que multiplicar el resto obtenido para la raíz cuadrada recíproca por X. Al igual que para la división, la única manera de obtener la raíz cuadrada es obtener su recíproco y multiplicarlo por X.

$$rem_i = X \cdot [1 + d_{i-1}^2 \cdot (rem_{i-1} - 1)] \tag{4.26}$$

Una vez que se han obtenido las expresiones alternativas para calcular el resto, hay dos aspectos importantes que deben ser mencionados. Por un lado, el resto obtenido de estas expresiones es el resto del resultado tal y como se obtiene del algoritmo (en el redondeo tradicional antes de calcular el resto del resultado se le suma una constante y se trunca a una longitud determinada). Por lo tanto, el método clásico de redondeo descrito en 1.3 no puede ser usado aquí. Se hace necesario el diseño de un nuevo método de redondeo que use el resultado del algoritmo sin ninguna transformación previa. Esto se explicará en secciones siguientes. Por otra parte, es necesario comprobar si estas expresiones producen el mismo resultado que el cálculo tradicional del resto o no. Como se explicará, más tarde estas expresiones serán validadas por medio de simulaciones exhaustivas.

4.3. Implementación del cálculo del resto en paralelo

Es necesario realizar algunas consideraciones acerca de la implementación en hardware del cálculo en paralelo del resto. En primer lugar, es importante destacar que las expresiones obtenidas en el apartado anterior no calculan el resto de ω' (ω' es, como se mencionó en el capítulo 1, el resultado del algoritmo

$\hat{\omega}$ despues de sumarle una constante y truncarlo a $n+1$ bits). Estas ecuaciones proporcionan el resto de $\hat{\omega}$.

Por otra parte, la precisión finita disponible en hardware hace que los resultado obtenidos no sean exactos. Este hecho hace que sea necesario establecer un orden para realizar las distintas operaciones para llegar a un resultado. Esto es así porque, al tener una precisión finita, el orden de las operaciones afecta al resultado final obtenido. De este modo, se puede intuir que el resultado obtenido en paralelo para el resto, no coincida en todas las ocasiones con el calculado de manera tradicional. Además, como se verá a continuación, al establecer un orden, será posible hacer una estimación del error que afecta al cálculo paralelo del resto (con respecto del cálculo tradicional).

Por último, señalar que el cálculo en paralelo del resto requiere un multiplicador adicional para no bloquear la ejecución del algoritmo. La disponibilidad de este multiplicador hace que el cálculo en paralelo no tenga ningún efecto, en términos de tiempo de ejecución, en la ejecución del algoritmo.

Lo que se expone a continuación, es el orden lógico en que se deben realizar las distintas operaciones que componen el cálculo del resto paralelo para cada una de las operaciones. Una vez establecido el orden, se describe el proceso de obtención de una estimación del error del resto calculado en paralelo (este error se definirá como la diferencia entre el resto obtenido de manera tradicional y el resto obtenido en paralelo con la ejecución del algoritmo).

4.3.1. Obtención del resto del recíproco y la división para el algoritmo GLD

En el caso del recíproco la operación que es necesario implementar es la expresión para el resto descrita en la ecuación (4.5). Esta ecuación puede ser reescrita del siguiente modo:

$$rem_i = 1 - K_i \cdot (1 - rem_{i-1}) \tag{4.27}$$

Esta expresión se compone de tres operaciones aritméticas que hay que realizar en un determinado orden. A saber:

- La primera operación es el complemento a 1 de rem_{i-1} que será representado como:

$$\overline{rem_{i-1}} = 1 - rem_{i-1} \tag{4.28}$$

- La segunda operación es una multiplicación. Como es habitual cuando se calcula el resto para llevar a cabo el redondeo, es necesario utilizar todos los bits obtenidos en el multiplicador. Es decir, el resultado del multiplicador no puede ser redondeado ni truncado.

$$K_i \cdot \overline{rem_{i-1}} \tag{4.29}$$

- La tercera operación es otro complemento a 1.

$$\overline{K_i \cdot \overline{rem_{i-1}}} \tag{4.30}$$

  Esta es la expresión final para el resto paralelo para el recíproco, que contiene tres operaciones aritméticas realizadas en el orden anterior. Si se quisiera obtener el resto para la división, habría una operación más. Una multiplicación por el dividendo Y.

Sin embargo, la expresión final para el cálculo del resto de manera convencional viene dado por la siguiente expresión:

$$rem_i = \overline{\hat{\omega} \cdot X} \tag{4.31}$$

Evidentemente, aunque las expresiones (4.30) y (4.31) están diseñadas para obtener la misma magnitud, la secuencia de operaciones aritméticas utilizadas es diferente en cada caso. Esto hace, dado que en las implementaciones hardware el orden y el tipo de operación tiene efecto sobre el resultado, estas expresiones no produzcan siempre el mismo resultado.

Error del resto paralelo: El valor que realmente se necesita para hacer el redondeo es el de la ecuación (4.31). Pero el que realmente se utiliza, es el que se ha calculado en paralelo (ecuación (4.30)). Para este análisis denominaremos $Trem_i$ al valor del resto calculado de manera tradicional y $Prem_i$ al resto calculado en paralelo. Teniendo en cuenta esto, definimos el error del resto paralelo (a partir de ahora ε_{par}) como la diferencia entre ambos restos.

$$\varepsilon_{par} = Trem_i - Prem_i \tag{4.32}$$

Para realizar este análisis del error del resto se asume que el resto de la iteración anterior está calculado utilizando todos los productos parciales del multiplicador, que no se considera ningún error en los operandos y que tampoco lo hacen las operaciones de complementado a 1. Hay que tener en cuenta,

también, que para las multiplicaciones que intervienen en el cálculo del resto se utilizan todos los productos parciales. Es decir, el resultado obtenido en el multiplicador no está redondeado ni truncado.

En primer lugar, será necesario obtener una expresión expandida con todas las contribuciones al error de $Trem_i$.

$$Trem_i = 1 - X \cdot (\omega + \varepsilon_i) + \varepsilon^t_{Trem_i} \tag{4.33}$$

Es posible simplificar esta expresión, de modo que se obtiene una expresión final para el resto calculado de manera convencional.

$$Trem_i = -X \cdot \varepsilon_i + \varepsilon^t_{Trem_i} \tag{4.34}$$

Por otro lado, se debe de hacer lo mismo para el resto calculado en paralelo. Se parte de la ecuación (4.5) incluyendo todas las contribuciones al error.

$$Prem_i = 1 - \hat{K}_i \cdot (1 - Trem_{i-1}) + \varepsilon_{Prem_i} \tag{4.35}$$

En esta expresión, $Trem_{i-1}$ se puede expandir tal y como se explicó anteriormente. Quedan por ver las contribuciones al error de $\hat{K}_i$. Son las siguientes (ver sección 2.2):

$$\hat{K}_i = 2 - X \cdot \hat{\omega}_{i-1} - \varepsilon^t_{r_{i-1}} + \varepsilon^t_{K_i} \tag{4.36}$$

Expandiendo un poco más esta expresión se obtiene:

$$\hat{K}_i = 2 - X \cdot \omega - X \cdot \varepsilon_{i-1} - \varepsilon^t_{r_{i-1}} + \varepsilon^t_{K_i} \tag{4.37}$$

Esta expresión se sustituye en la ecuación (4.35). Expandiéndola y eliminando un término despreciable se obtiene la expresión para el resto calculado en paralelo.

$$Prem_i = -X \cdot \varepsilon_i + \varepsilon^t_{Trem_i} - X \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{K_i} - \varepsilon^t_{r_{i-1}} + \varepsilon^t_{Trem_{i-1}}) + \varepsilon^t_{Prem_i} \tag{4.38}$$

Finalmente, calculando, según la expresión (4.32), la diferencia entre las ecuaciones (4.34) y (4.38) se obtiene el error del resto en paralelo.

$$\varepsilon_{par} \simeq X \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{K_i} - \varepsilon^t_{r_{i-1}} + \varepsilon^t_{Trem_{i-1}}) - \varepsilon^t_{Prem_i} \tag{4.39}$$

Esta expresión será validada más adelante, a la vista de los resultados de las simulaciones exhaustivas que se presentarán en las secciones siguientes. Tal

y como viene siendo usual en todos los análisis de errores presentados, la estimación de este error para la división se obtiene mediante la multiplicación de la misma expresión por el dividendo Y.

4.3.2. Obtención del resto de la raíz cuadrada recíproca y la raíz cuadrada para el algoritmo GLD

El proceso a seguir para la raíz cuadrada recíproca es análogo al seguido para el caso del recíproco. Se parte de la expresión obtenida para el cálculo del resto en paralelo (ecuación (4.11)). Esta ecuación puede ser reescrita del siguiente modo:

$$rem_i = 1 - K_i^2 \cdot (1 - rem_{i-1}) \tag{4.40}$$

A continuación realizamos una descripción de las operaciones aritméticas que se requieren para implementar la expresión anterior.

- La primera de las operaciones aritméticas consiste en una multiplicación. Hay que elevar al cuadrado la magnitud K_i.

$$K_i^2 \tag{4.41}$$

- La siguiente operación consiste en el complemento a uno del resto de la iteración anterior.

$$\overline{rem_{i-1}} \tag{4.42}$$

- La tercera operación consiste en realizar la otra multiplicación que aparece en la expresión (4.38). Como ya se ha mencionado en alguna ocasión el resultado de esta multiplicación no debe de ser redondeado.

$$K_i^2 \cdot \overline{rem_{i-1}} \tag{4.43}$$

- La última de las operaciones es un complemento a 1.

$$\overline{K_i^2 \cdot \overline{rem_{i-1}}} \tag{4.44}$$

La consideración que es necesario hacer aquí es la misma que para el caso del recíproco. Las dos expresiones (la tradicional y la del cálculo en paralelo) para el cálculo del resto, realizan operaciones aritméticas distintas. Por lo tanto, no siempre van a producir el mismo resultado.

Error del resto paralelo: Queda por establecer para la ecuación (4.38) el error del resultado que produce, derivado de la implementación hardware de las operaciones aritméticas que lo componen. Para ello partimos de las mismas asunciones genéricas que se realizaron en el caso del recíproco. Igual que en el caso anterior, el error del resto calculado en paralelo viene dado por la diferencia entre el resto calculado de manera convencional y el propio resto paralelo.

$$\varepsilon_{par} = Trem_i - Prem_i \tag{4.45}$$

Por un lado, es necesario explicitar las distintas contribuciones al error del resto calculado convencionalmente. Son las siguientes:

$$Trem_i = 1 - X \cdot \hat{\omega}_i^2 + \varepsilon^t_{Trem_i} \tag{4.46}$$

Esta ecuación puede ser expandida hasta obtener:

$$Trem_i = -\frac{2}{\omega} \cdot \varepsilon_i - \frac{1}{\omega^2} \cdot \varepsilon_i^2 + \varepsilon^t_{Trem_i} \tag{4.47}$$

Por otro lado, hay que hacer lo mismo para el cálculo paralelo del resto. La expresión de este en una implementación hardware tiene la siguiente forma:

$$Prem_i = 1 - \hat{K}_i^2 \cdot (1 - Trem_{i-1}) + \varepsilon^t_{Prem_i} \tag{4.48}$$

Es necesario, para poder obtener la formula final expandida, analizar las contribuciones de K_i^2. En primer lugar, para obtener $\hat{K}$ hay que utilizar la fórmula:

$$\hat{K}_i = \frac{1}{2} \cdot (3 - \hat{r}_{i-1}) + \varepsilon^t_{K_i} \tag{4.49}$$

Es necesario también tener en cuenta, la composición de $\hat{r}_{i-1}$.

$$\hat{r}_{i-1} = X \cdot (\omega^2 + \varepsilon_{i-1}^2 + 2 \cdot \omega \cdot \varepsilon_{i-1} + \varepsilon^t_{\omega^2_{i-1}}) + \varepsilon^t_{r_{i-1}} \tag{4.50}$$

Se incluyen todas estas expresiones en la ecuación (4.48). De manera análoga al recíproco, restando la expresión para $Prem_i$ de $Trem_i$ y eliminando algunos términos despreciables se llega a la siguiente expresión para el resto calculado en paralelo

$$\varepsilon_{par} \simeq \varepsilon_{i-1} \cdot (\frac{\varepsilon^t_{\omega^2_{i-1}}}{\omega^3} + \frac{\varepsilon^t_{r_{i-1}}}{\omega} + \varepsilon^t_{K_i} + \varepsilon^t_{Trem_i}) - \varepsilon^t_{Prem_i} \tag{4.51}$$

La correspondiente expresión para la raíz cuadrada se obtiene multiplicando la expresión anterior por el operando X.

4.3.3. Obtención del resto del recíproco y la división para el algoritmo NR

Se puede repetir todo el mismo proceso, pero ahora aplicado al algoritmo NR. Describimos en primer lugar las operaciones para el caso del recíproco. La ecuación definida en (4.19) para el cálculo en paralelo del resto para el recíproco, puede ser reescrita del siguiente modo:

$$rem_i = 1 - d_{i-1} \cdot (1 - rem_{i-1}) \tag{4.52}$$

A continuación hacemos un listado de las operaciones aritméticas que componen la expresión anterior:

- Complemento a 1 de rem_{i-1}.

$$\overline{rem_{i-1}} = 1 - rem_{i-1} \tag{4.53}$$

- La siguiente operación es una multiplicación:

$$K_i \cdot \overline{rem_{i-1}} \tag{4.54}$$

- La última operación consiste en el complemento a 1 del resultado de la multiplicación anterior

$$\overline{K_i \cdot \overline{rem_{i-1}}} \tag{4.55}$$

Error del resto paralelo: Una vez conocidas las operaciones aritméticas y el orden en que se realizan, hacemos una estimación del error de este cálculo. No hay diferencias significativas con el procedimiento establecido para el algoritmo GLD.

En primer lugar, la expresión final del resto calculado de manera tradicional tiene la siguiente forma:

$$Trem_i = -X \cdot \varepsilon_i + \varepsilon^t_{Trem_i} \tag{4.56}$$

Por otra parte, siguiendo el mismo proceso que en el caso del algoritmo GLD, se obtiene la expresión para el resto calculado en paralelo.

$$Prem_i = -X \cdot \varepsilon_i - \varepsilon^t_{Trem_i} - X \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{d_{i-1}} - \varepsilon^t_{n_{i-1}} + \varepsilon^t_{Trem_{i-1}}) + \varepsilon^t_{Prem_i} \tag{4.57}$$

A través de las dos ecuaciones anteriores se llega a la expresión final del error para el resto paralelo.

$$\varepsilon_{par} = -X \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{d_{i-1}} - \varepsilon^t_{n_{i-1}} + \varepsilon^t_{Trem_{i-1}}) - \varepsilon^t_{Prem_i} \tag{4.58}$$

4.3.4. Obtención del resto de la raíz cuadrada recíproca y la raíz cuadrada para el algoritmo NR

Igual que antes se parte de de la expresión del calculo en paralelo del resto para la raíz cuadrada recíproca, que viene dado por:

$$rem_i = 1 - d^2_{i-1} \cdot (1 - rem_{i-1}) \tag{4.59}$$

Las operaciones aritméticas a realizar son las siguientes:

- La primera operación consiste en la multiplicación necesaria para elevar al cuadrado d_{i-1}.
$$d^2_{i-1} \tag{4.60}$$
- La segunda de ellas consiste en un complemento a 1:
$$\overline{rem_{i-1}} = 1 - rem_{i-1} \tag{4.61}$$
- La siguiente corresponde a la multiplicación:
$$d^2_{i-1} \cdot \overline{rem_{i-1}} \tag{4.62}$$
- La última es otro complemento a 1.
$$\overline{d^2_{i-1} \cdot \overline{rem_{i-1}}} \tag{4.63}$$

Error del resto paralelo: Las expresiones que hay que utilizar, al igual que en todos los casos anteriores, son el resto calculado de manera tradicional:

$$Trem_i = -\frac{2}{\omega} \cdot \varepsilon_i - \frac{1}{\omega^2} \cdot \varepsilon^2_i + \varepsilon^t_{Trem_i} \tag{4.64}$$

El resto calculado en paralelo, supone el desarrollo de todas las contribuciones al error de la expresión:

$$Prem_i = 1 - \hat{d}^2_i \cdot (1 - Trem^t_{i-1}) + Trem^t_{i-1} \tag{4.65}$$

Desarrollando la expresión anterior, al igual que en todos los casos anteriores, y eliminando algunos términos despreciables se llega a una expresión aproximada para el error del resto calculado en paralelo.

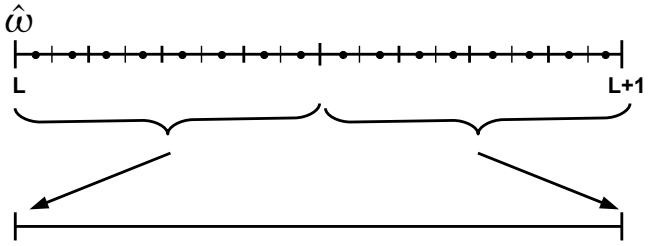


Figura 4.1: Nuevo esquema de redondeo al más próximo

$$\varepsilon_{par} = \varepsilon_{i-1} \cdot (\frac{\varepsilon^t_{s_{i-1}}}{\omega^3} - \frac{\varepsilon^t_{n_{i-1}}}{\omega} + \varepsilon^t_{d_{i-1}} + \varepsilon_{Trem^t_{i-1}}) + \varepsilon_{Prem^t_{i-1}} \tag{4.66}$$

4.4. Nuevo método de redondeo

La manera alternativa de obtener el resto, descrito en la sección anterior, hace necesario el diseño de un método de redondeo para algoritmos multiplicativos distinto del tradicional, descrito en la sección 1.3. También se ha mencionado que este nuevo método debe de estar basado en el resultado obtenido del algoritmo de manera directa ($\hat{\omega}$), en lugar de estar basado en una versión transformada del resultado (ω'). Esto es debido a que el resto del que se dispone es, precisamente, el resto de $\hat{\omega}$.

En la figura 4.1 se muestra un esquema del nuevo método de redondeo para el caso concreto de redondeo al más próximo. La parte superior de la figura representa el intervalo de posibles valores que puede tomar el resultado del algoritmo $\hat{\omega}$ entre dos puntos representables con n bits de precisión (L y $L+1$). Está también representado el punto medio que, para ser representado, requerirá $n+1$ bits de precisión. La parte inferior representa la recta que contiene los dos posibles valores finales del resultado después de ser redondeado. Estos, debido a los requerimientos del estándar, sólo pueden ser, puntos representables de precisión y exactitud n.

Como en cualquier método de redondeo, según el estándar IEEE 754, el objetivo que se persigue es obtener el mismo resultado que si se hubiera redondeado

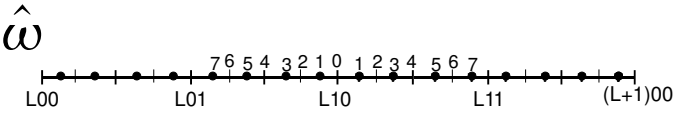


Figura 4.2: Casos que necesitan el cálculo del resto

el resultado exacto. Supongamos que el resultado obtenido del algoritmo ($\hat{\omega}$) es alguno de los puntos entre los dos puntos representables entre el punto L y el punto $L+1$. Supondremos además que tiene una exactitud mayor o igual que $2^{-(n+2)}$ bits, es decir, $|\omega - \hat{\omega}| \leq 2^{-(n+2)}$.

La acción de redondeo al más próximo es equivalente a decir que el resultado final será bien el punto de precisión n más cercano al resultado exacto. Es decir, el resultado final depende de donde está el resultado exacto. Entonces, si el resultado exacto está a la izquierda del punto medio entre los dos puntos representables L y $L+1$ la acción de redondeo será truncar. Si el resultado exacto está a la derecha, la acción necesaria será incrementar. El caso para el cual el resultado exacto es, justamente, el punto medio entre L y $L+1$ nunca se puede producir [OF97].

En el método tradicional, para conocer la posición del resultado exacto, se utiliza una aproximación al resultado (ω'). Esta aproximación, tal y como se vió en la sección 1.3, se obtiene transformando $\hat{\omega}$, de tal manera que tiene una exactitud mayor o igual $2^{-(n+1)}$. De este modo es necesario conocer el signo del resto para determinar en que mitad de la recta se encuentra el resultado exacto. Su posición determina la acción de redondeo que es necesario tomar.

En este nuevo método de redondeo, es necesario cambiar todo el proceso descrito en el párrafo anterior porque disponemos del resto de $\hat{\omega}$ no del resto de ω'. Sin pérdida de generalidad, suponemos que $\hat{\omega}$ se calcula con una exactitud de $2^{-(n+2)}$ y con una precisión de $n+5$ bits.

En la figura 4.2 se representan todos los valores que puede tomar $\hat{\omega}$ entre dos puntos de precisión n que se diferencian en 1 ulp. Los puntos etiquetados se representan con tres bits por simplicidad de la figura. L es el bit en la posición n, luego habría que añadirle tres bits más. Por ejemplo, la posición $L10$ es el número $1.xx....xL10000$. Hay 8 posibles valores entre $L10$ y $L11$ que son de la forma $1.xx....xL10xxx$, desde $1.xx....xL10000$ a $1.xx....xL10111$.

Teniendo en cuenta los requerimientos de exactitud para $\hat{\omega}$, los puntos de la figura a la izquierda de $L01$ no necesitan el cálculo del resto, puesto que se sabe

que el resultado exacto siempre estará por debajo de $L10$. Esto es debido a la exactitud requerida para el resultado que es:

$$|\omega - \hat{\omega}| \leq 2^{-(n+2)} \tag{4.67}$$

Luego, la acción de redondeo que se necesita para estos puntos siempre es truncar.

Los mismos argumentos pueden ser aplicados a todos aquellos puntos que en la figura están a la derecha de $L11$. En este caso, la acción es siempre incrementar. Esto es así, porque, si la exactitud es mayor o igual que $2^{-(n+2)}$, el resultado exacto siempre será mayor que el punto $L10$.

El resto de puntos, que están etiquetados con un número en la figura 4.2, son los que necesitan del cálculo del resto. Cada una de las etiquetas asignadas a los distintos puntos representables para $\hat{\omega}$, representa la distancia de dicho punto al punto medio $L10$ (medida en número de intervalos de $2^{-(n+5)}$). La distancia se obtiene tal y como se explica a continuación. Por ejemplo, si escogemos el punto 7, la distancia de este punto al punto medio $L10$ es $7 \cdot 2^{-(n+5)}$. Al igual que en la figura anterior, en esta figura no se representan todos los bits de cada punto. Por ejemplo, $L10$ que identifica al punto medio, son los bits n, $n+1$ y $n+2$ de dicho punto. Así, este punto será $1.xxxxx...xL10000$, el inmediatamente inferior es $1.xxxxx...xL01111$, el inmediatamente superior $1.xxxxx...xL10001$ y así sucesivamente. De este modo, el punto etiquetado como 7 es el punto $1.xx....xL01001$, si está a la izquierda de $L10$, y $1.xx....xL10111$ si está a la derecha.

Para saber si el resultado exacto está a la izquierda o a la derecha del punto medio, utilizaremos la distancia entre el resultado real y la aproximación $\hat{\omega}$ que obtenemos del algoritmo. La distancia $\omega - \hat{\omega}$ permite saber si el punto está a la izquierda o a la derecha de $L10$. El objetivo es poder relacionar esta distancia con el valor del resto. De este modo en función del valor del resto se podrá obtener esta información.

Este proceso es análogo para todos los puntos, aunque el valor que permite distinguir la posición del resultado exacto depende de los cinco últimos bits de cada punto. Veamos, por ejemplo, el caso de todos aquellos puntos que están justo a la izquierda de $L10$, es decir, que tienen la terminación $L01111$. El valor de la distancia, mayor o menor que $2^{-(n+5)}$, permite discriminar la posición del resultado exacto respecto del punto $L10$.

$$\begin{aligned} \omega - \hat{\omega} &> 2^{-(n+5)} \Rightarrow \omega > L10 \\ \omega - \hat{\omega} &< 2^{-(n+5)} \Rightarrow \omega < L10 \end{aligned} \tag{4.68}$$

No se considera el caso $\omega - \hat{\omega} = 2^{-(n+5)}$ porque esto significaría que el resultado exacto es justo el punto medio. Se ha determinado que este caso no se puede dar en algoritmos multiplicativos [OF97]. Sin embargo este punto, es el que se utiliza para determinar si el resultado exacto está a la izquierda o a la derecha de $L10$.

$$\hat{\omega} = \omega - 2^{-(n+5)} \tag{4.69}$$

Hay que obtener el resto para este valor de $\hat{\omega}$. Se obtiene el mismo valor del resto tanto para el recíproco como para la raíz cuadrada recíproca. En el caso del recíproco de manera exacta y en el de la raíz cuadrada recíproca eliminando algún término despreciable. De esto modo el proceso de redondeo es común a ambas operaciones. El valor obtenido del resto es el siguiente:

$$rem = X \cdot 2^{-(n+5)} \tag{4.70}$$

Teniendo en cuenta las ecuaciones (4.68) y (4.70), se establece un criterio que predice donde está el resultado exacto en función del valor del resto

$$\begin{aligned} rem &> X \cdot 2^{-(n+5)} \Rightarrow \omega > L10 \\ rem &< X \cdot 2^{-(n+5)} \Rightarrow \omega < L10 \end{aligned}$$

Este es el criterio que hay que aplicar para todos aquellos puntos con la terminación $L01111$. Se puede aplicar el mismo proceso a todos aquellos puntos con otras terminaciones que necesitan del cálculo del resto. Lo único que cambia es el valor con el que hay que comparar el resto para determinar donde se encuentra el resultado exacto. De este modo, es posible construir una nueva tabla de redondeo que está basada en el valor de $\hat{\omega}$ (Tabla 4.1). Esta tabla es la que define por completo el nuevo método de redondeo propuesto, que consiste en la determinación de la acción de redondeo en base al resultado de la comparación del resto con una serie de valores.

Esta tabla de redondeo y la nueva manera de calcular el resto en paralelo han sido validados por medio de simulaciones exhaustivas. Los resultados de estas validaciones se presentan en la siguiente sección.

4.5. Validación del método

Esta sección está dedicada a explicar como se construyeron y utilizaron las simulaciones exhaustivas para la validación del nuevo método de redondeo. Asi-

$\omega[n+1:n+5]$	Condición	Acción
01001	$rem > 7 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < 7 \cdot X \cdot 2^{-(n+5)}$	TRUNC
01010	$rem > 6 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < 6 \cdot X \cdot 2^{-(n+5)}$	TRUNC
01011	$rem > 5 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < 5 \cdot X \cdot 2^{-(n+5)}$	TRUNC
01100	$rem > 4 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < 4 \cdot X \cdot 2^{-(n+5)}$	TRUNC
01101	$rem > 3 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < 3 \cdot X \cdot 2^{-(n+5)}$	TRUNC
01110	$rem > 2 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < 2 \cdot X \cdot 2^{-(n+5)}$	TRUNC
01111	$rem > X \cdot 2^{-(n+5)}$	INC
	$rem < X \cdot 2^{-(n+5)}$	TRUNC
10000	$rem > 0$	INC
	$rem < 0$	TRUNC
10001	$rem > -X \cdot 2^{-(n+5)}$	INC
	$rem < -X \cdot 2^{-(n+5)}$	TRUNC
10010	$rem > -2 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < -2 \cdot X \cdot 2^{-(n+5)}$	TRUNC
10011	$rem > -3 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < -3 \cdot X \cdot 2^{-(n+5)}$	TRUNC
10100	$rem > -4 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < -4 \cdot X \cdot 2^{-(n+5)}$	TRUNC
10101	$rem > -5 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < -5 \cdot X \cdot 2^{-(n+5)}$	TRUNC
10110	$rem > -6 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < -6 \cdot X \cdot 2^{-(n+5)}$	TRUNC
10111	$rem > -7 \cdot X \cdot 2^{-(n+5)}$	INC
	$rem < -7 \cdot X \cdot 2^{-(n+5)}$	TRUNC

Tabla 4.1: Tabla para redondeo al más próximo

mismo se presentan los resultado obtenidos que servirán para la evaluación del método y su comparación con otros ya existentes. Todas las simulaciones presentadas fueron realizadas con Maple [GLM08]. El propósito de estas simulaciones

	Recip.	Raíz Recip.
GLD	0.8 %	1.6 %
NR	0.8 %	1.6 %

Tabla 4.2: Porcentaje de casos, respecto del total, donde la acción de redondeo no es correcta

fue corroborar el resultado teórico de que el resto calculado en paralelo no produce el mismo resultado que el resto calculado de manera tradicional en todas las ocasiones. Además, se quería determinar en cuantas ocasiones respecto del total era necesario, entonces, el cálculo tradicional del resto. Este resultado permite proponer un nuevo algoritmo de latencia variable para el redondeo.

Todas las simulaciones se realizaron tanto para el algoritmo NR como para el algoritmo GLD. Se diseño un código que reproduce la ejecución en hardware del algoritmo. Se simuló la contribución que la ejecución hardware de las operaciones aritméticas introduce en el error del resultado. Sin pérdida de generalidad, las operaciones simuladas son el recíproco y la raíz cuadrada recíproca. Se implementaron tanto el método clásico de redondeo como el nuevo método.

El formato de datos escogidos para las simulaciones fue el de precisión simple, puesto que otros formatos de mayor precisión consumen muchos más recursos computacionales. Para obtener resultados acerca del redondeo bastó con simular una única iteración de los algoritmos. Al igual que en la construcción de modelo teórico del nuevo método, la precisión escogida para las operaciones intermedias fue de $n + 5$ bits.

El primer y más importante resultado que se extrae de las simulaciones es que el valor obtenido en paralelo del resto no es válido para todos los casos. Este resultado ya se había previsto en el desarrollo teórico del método. Los números concretos de este resultado se muestran en la tabla 4.2. A pesar de esto, el porcentaje de casos en que el valor del resto obtenido no es adecuado es muy pequeño para cualquiera de los casos simulados.

Como ya se había mencionado en secciones anteriores, esto es debido a problemas en la implementación del cálculo del resto. Este problema está provocado por que la representación de los números en hardware hace que las operaciones aritméticas pierdan la propiedad asociativa. Por lo tanto, operaciones aritméticas distintas que llegan al mismo resultado de manera teórica no lo hacen cuando son implementadas en hardware. El resultado es que, cuando se calcula el resto en paralelo, se obtengan resultados ligeramente diferentes.

	GLD		NR	
$\hat{\omega}[n+1:n+5]$	Recip.	Raíz Recip.	Recip.	Raíz Recip.
01101	–	0.8 %	–	0.8 %
01110	–	9.0 %	–	9.0 %
01111	2.1 %	21.9 %	2.1 %	21.9 %
10000	17.6 %	14.3 %	17.6 %	14.3 %
10001	4.8 %	5.1 %	4.8 %	5.1 %
10010	–	0.2 %	–	0.2 %

Tabla 4.3: Porcentaje de casos con los mismos últimos 5 bits (respecto del total con la misma terminación) con un valor del resto paralelo incorrecto.

4.5.1. Algoritmo de latencia variable

A la vista de estos primeros resultados, queda claro que, para el esquema propuesto, no es posible proponer un método de redondeo de los resultados de algoritmos multiplicativos donde se evite el cálculo del resto para todos los casos. Sin embargo, es perfectamente factible proponer un algoritmo de redondeo de latencia variable con muy buenos porcentajes de casos en los que se evita el cálculo del resto. El esquema final, calculará el resto en paralelo en la gran mayoría de las ocasiones y, en algunos casos, habrá que recurrir al modo convencional de calcular el resto.

Teniendo en cuenta los resultados obtenidos hasta este punto se realizaron simulaciones adicionales. El siguiente paso era confirmar, según lo explicado en la sección anterior, exactamente qué casos de $\hat{\omega}$ necesitaban del cálculo convencional del resto y para cuales bastaba con la obtención en paralelo del mismo. Dichos casos fueron clasificados en base a sus últimos 5 bits. Lo que finalmente se calculó fue qué porcentaje de casos con la misma terminación de 5 bits, respecto del total de casos que terminan con los mismos últimos 5 bits, necesitan del cálculo convencional del resto.

Como cabía esperar, para todos los valores de $\hat{\omega}$ con los mismos 5 últimos bits, no siempre se necesita el cálculo convencional del resto. Solo un porcentaje pequeño de estos casos lo necesita (Tabla 4.3). Para cualquiera de los dos algoritmos considerados (NR y GLD), solo tres combinaciones de los últimos 5 bits entre todas las posibles, necesitan el cálculo del resto de manera convencional en algunas ocasiones. Por ejemplo, si los últimos 5 bits de $\hat{\omega}$ son 10000 para el casos del recíproco, el cálculo del resto en paralelo es correcto en el 82,74 % de las veces que se produce esa combinación.

	GLD		NR	
$\hat{\omega}[n+1:n+5]$	Recip.	Raíz Recip.	Recip.	Raíz Recip.
01101	–	3.13 %	–	3.13 %
01110	–	3.13 %	–	3.13 %
01111	3.12 %	3.12 %	3.12 %	3.12 %
10000	3.14 %	3.12 %	3.14 %	3.12 %
10001	3.11 %	3.12 %	3.11 %	3.12 %
10010	–	3.12 %	–	3.12 %
Total	9.37 %	18.74 %	9.37	18.74 %
Otros [Sch95]	25.01 %	25.01 %	25.01	25.01 %
Otros works[Obe99]	12.49 %	12.49 %	12.49	12.49 %

Tabla 4.4: Porcentaje de casos con resto paralelo incorrecto respecto del total

En el caso de la raíz cuadrada recíproca, el número de combinaciones de los últimos 5 bits donde se producen valores erróneos del resto son 6. Las combinaciones 01010 y 10010 fallan en la predicción en un número muy reducido de casos (0,8 % y 0,2 % respectivamente). Este número de casos es lo suficientemente pequeño como para que estos casos puedan ser almacenados en una tabla muy pequeña. De esta manera es posible reducir el número de combinaciones problemáticas a 4. Sin embargo, el número total de casos para el resto de las combinaciones, tanto en el caso del recíproco como de la raíz cuadrada recíproca, es demasiado grande. Esto provoca que no sea posible predecir, para una combinación de 5 últimos bits, en que casos el valor del resto obtenido en paralelo es adecuado o no. Se consideró la posibilidad de hacerlo en una tabla o sintetizando una función lógica, pero el número de casos es demasiado grande.

Es necesario destacar, también, que los resultados obtenidos en estas simulaciones son coherentes con las estimaciones teóricas obtenidas para el error del cálculo del resto en paralelo. La distancia de los puntos para los que en algunos casos no se predice bien la acción de redondeo al punto medio $L10000$ es menor o igual que el valor de dicho error.

$$\varepsilon_{par} \geq 3 \cdot 2^{-(n+5)} \tag{4.71}$$

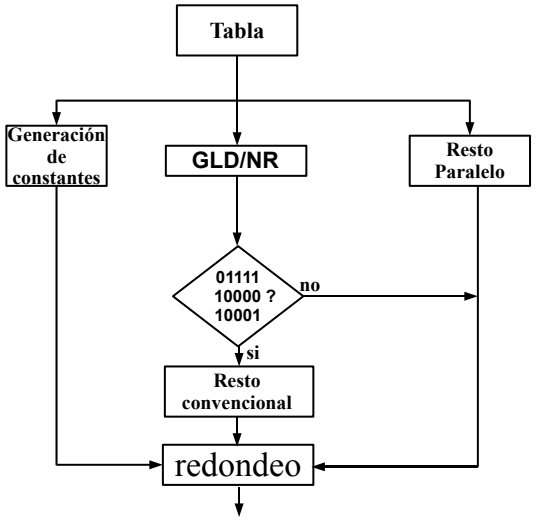
Por esta razón, el resto calculado en paralelo no distingue en ocasiones de manera adecuada la posición del resultado exacto. Este error se obtiene de la evaluación concreta de las expresiones (4.39), (4.51), 4.58 y 4.66. Esto hace que para estos puntos, el valor del resto no tenga la exactitud necesaria para poder

discernir si el resultado real está por encima o por debajo del punto $L10000$.

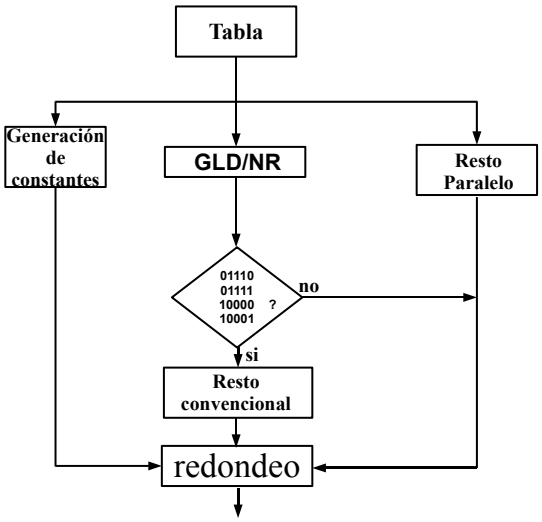
La opción factible, a la vista de los resultados, para construir un método de latencia variable es realizar el cálculo paralelo del resto en todos los casos excepto cuando se produce una de las combinaciones que en algunos casos no predicen bien la acción de redondeo. Cuando aparece una de las combinaciones de la tabla 4.3 se realizará el cálculo convencional del resto. Se excluirán las dos combinaciones 01010 y 10010 para la raíz cuadrada recíproco porque, como habíamos dicho, estos casos pueden ser almacenados en una tabla de tamaño muy pequeño para ser detectados.

4.5.2. Eficiencia del método de latencia variable

Una vez que se decidió el esquema necesario para implementar un método de latencia variable, se realizaron simulaciones para determinar el porcentaje de casos total en que hay que recurrir al cálculo convencional del resto. Es decir, para medir la eficiencia del método. Se midió el porcentaje de casos respecto del total, para cada una de las posibles combinaciones de los últimos 5 bits. La suma de todos estos porcentajes proporciona el porcentaje de casos respecto del total que necesitan del cálculo convencional del resto. Los resultados de estas simulaciones se encuentra en la tabla 4.4. En la tabla es también posible ver el porcentaje total de ocasiones en que se necesita el cálculo del resto convencional para cada operación. Este porcentaje es de un 9,37 % para el caso de raíz cuadrada recíproca y de 12,49 % para la raíz cuadrada recíproca. En la parte de inferior de la tabla, también se incluyen los resultados de otros trabajos realizados previamente para posibilitar la comparación. Estos porcentajes fueron obtenidos realizando las correspondientes simulaciones de estos métodos con las mismas condiciones que para el método propuesto aquí. El objetivo de estas simulaciones era obtener el número exacto de casos, respecto del total, en que se necesita el cálculo convencional del resto. Esto es así porque, el resultado que estos trabajos proporcionaban originalmente era la fracción de posibles entradas de la tabla de redondo que necesitaban del cálculo convencional del resto. Como se puede ver el método mejora los resultados para división con respecto a los otros dos métodos comparados. En el caso de la raíz cuadrada únicamente mejora los resultados respecto a uno de los métodos. Sin embargo tiene la ventaja de ser un método unificado para la raíz cuadrada, la división y sus recíprocos.



(a) Recíproco



(b) Raíz cuadrada recíproca

Figura 4.3: Esquema propuesto para las unidades con cálculo del resto en paralelo

4.6. Método propuesto

A la vista de los resultados expuestos en la sección anterior, se propone un esquema completo para la implementación del método propuesto (Figura 4.3). Este permitirá el cálculo en paralelo del resto en la gran mayoría de los casos. Este cálculo en paralelo se realizará de acuerdo con las expresiones propuestas en secciones anteriores. Será necesario, además, generar todas las constantes necesarias para la comparación con el redondeo. Este proceso también se realiza en paralelo con la ejecución del algoritmo. Es necesario generar todas las constantes posibles (que aparecen en la tabla 4.1) puesto que, a priori, no se conoce cual de ellas será necesaria. Este cálculo consiste en desplazamientos del operando X. Después de calcularlas será necesario almacenarlas en una tabla en espera de que sean utilizadas. Esta tabla tendrá un tamaño pequeño de 14 entradas de 28 bits. Una vez que se conocen los últimos cinco bits del resultado $\hat{\omega}$, se escogerá la constante adecuada para ser comparada con el valor del resto. El resultado de esta comparación, que es muy rápida, determina la acción de redondeo. Esta constante podrá ser comparada con el resto calculado en paralelo o con el resto obtenido de manera convencional dependiendo del caso. La decisión de qué resto utilizar para realizar la comparación se toma una vez que se ha obtenido el resultado. Si los últimos 5 bits de este es alguna de las combinaciones de la tabla 4.4, será necesario proceder al cálculo del resto de manera convencional para, luego, proceder a su comparación con la constante adecuada. Si no es así, se utiliza el resto que había sido computado en paralelo. Por último, se realiza la comparación y se toma la acción según el resultado de esta.

Este método tiene utilidad si se utiliza un multiplicador para realizar el cómputo del algoritmo y se utiliza otro distinto para el cómputo del resto en paralelo. Dichos multiplicadores deben ser segmentados. Suponemos para ellos una latencia de 4 ciclos. Con esta configuración descrita, la latencia para el caso del recíproco, es de 8 ciclos. 12 para el caso en que es necesario el cálculo del resto de manera convencional. Finalmente, para la raíz cuadrada recíproca, el algoritmo necesita 13 ciclos cuando es suficiente con el cálculo del resto en paralelo y 17 si hay que calcular el resto de manera convencional.

4.7. Conclusión

En este capítulo se ha presentado un método que elimina por completo la latencia debida al cálculo del resto para la mayoría de los casos en el redondeo

de algoritmos multiplicativos. El método está basado en el cálculo en paralelo del resto y en un nuevo método de redondeo. Con estas mejoras, el número de veces en que hay que calcular el resto de manera convencional queda reducido a un 9 % de los casos en el casos del recíproco y la división. El porcentaje es de un 12 % en el caso de la raíz cuadrada recíproca y la raíz cuadrada. De este modo la arquitectura final obtenida es un método de latencia variable.

Se ha propuesto un método para calcular el resto en paralelo con la ejecución del algoritmo. Este método produce el resto del resultado directamente obtenido del algoritmo, sin ninguna transformación. Por esta razón, fue necesario presentar un nuevo método de redondeo a partir de esta aproximación al resultado sin transformar. El método consiste en comparar el resto con constantes fácilmente calculables. La constante utilizada depende de los últimos 5 bits del resultado obtenido.

El método propuesto ha sido validado por medio de simulaciones exhaustivas. Estas simulaciones proporcionan una medida exacta de que casos producen un valor adecuado del resto, calculándolo en paralelo, y cuales no. Los resultados extraídos de estas simulaciones muestran importantes reducciones en el número de veces que hay que calcular el resto de manera tradicional respecto de trabajos previos.

Capítulo 5

Método de redondeo con obtención simple del resto

En el capítulo anterior se explicó el diseño de un nuevo método de redondeo de latencia variable. Dicho método utiliza un modo de obtener el valor del resto, necesario para el redondeo, sin causar ninguna penalización adicional a la latencia del algoritmo. La manera de hacerlo era calcular dicho valor en paralelo. Esto se había conseguido a través de una expresión matemática que obtiene el resto en función del resto de la iteración anterior. Este método producía mejores resultados que trabajos previos comparables, pero tiene un inconveniente. Es necesario disponer de un multiplicador adicional para el cómputo del resto en paralelo. Lo que se propone en este capítulo es un método de redondeo de latencia variable que evita la necesidad del multiplicador adicional y que produce resultados prácticamente iguales que el método anterior. La ventaja es el ahorro de un multiplicador. Este método se basa en obtener el resto a través de una operación aritmética mucho más sencilla que la multiplicación, la resta. Este método sólo es válido para el algoritmo de Goldschmidt (GLD). No es posible proponer un método equivalente para el caso del algoritmo de Newton-Raphson. La propia estructura del algoritmo impide obtener el resto para la misma iteración que se está calculando. Por lo tanto este método únicamente será válido para el algoritmo GLD. En el caso de la división solamente se necesitará realizar el cálculo del resto de manera convencional en el 9 % de los casos. Este porcentaje será de un 12 % de los casos que mejora incluso los resultados del método anterior

5.1. Introducción

Ya se ha venido mencionando a lo largo de todo este trabajo que el estándar IEEE754 requiere que los resultados de las operaciones aritméticas estén redondeados según los modos descritos en dicho estándar. Todos los esfuerzos realizados hasta ahora por optimizar este proceso de redondeo han desembocado siempre en métodos de latencia variable. Es decir, métodos que realizan una acción u otra dependiendo del valor concreto del resultado. Básicamente, la acción será calcular el resto de manera convencional en unos casos y en otros, el resto no se calculará o se hará de manera alternativa.

Uno de los criterios que se deben tener en cuenta a la hora de diseñar métodos de redondeo mejorados es que su implementación ocupe la menor cantidad de área posible. El método propuesto en el capítulo anterior obtenía, en paralelo con la ejecución del algoritmo, el resto del resultado tal y como se obtiene directamente del algoritmo. ($\hat{\omega}$). Este cálculo en paralelo se obtiene a través del cómputo de una expresión matemática que, usando el resto de la iteración anterior, se calcula el de la siguiente. Aparte de restar de 1, la operación principal de esta expresión es la multiplicación en hardware. Al implementar este método es necesario disponer de un multiplicador adicional para no interferir en la ejecución del algoritmo. Esto supone un coste en hardware adicional.

Se presenta aquí un método similar al anterior, que proporciona resultados similares pero que no necesita de la multiplicación para obtener el resto en paralelo con la ejecución del algoritmo. La idea es que la magnitud r_i del algoritmo GLD, definida en la ecuación (1.44), es una magnitud muy parecida al valor del resto. Se obtiene el resto simplemente complementando dicha magnitud. El resto que se obtiene es el resto del resultado obtenido directamente del algoritmo sin ninguna modificación ($\hat{\omega}$). Esto permite, una vez que se ha obtenido el valor del resto, utilizar el mismo método de redondeo definido en la sección 4.4. Se obtiene así un método que en la mayoría de los casos no genera penalización en la latencia evitando el cálculo convencional del resto.

Debido a la estructura del algoritmo NR, no es posible establecer un método equivalente para él. El hecho de que las multiplicaciones no sean independientes entre sí para este algoritmo hace que no sea posible obtener el resto sin penalizar el tiempo de una multiplicación.

Al igual que en el método anterior, el cálculo del resto de modo convencional no se elimina completamente. El error inherente a la implementación hardware de las operaciones aritméticas hace que, en algunos casos, el resto no sea exactamente igual al convencional. Entonces, sólo es posible proponer un método de latencia variable.

Este método ha sido validado mediante simulaciones exhaustivas para obtener las medidas acerca de que porcentaje de casos, dependiendo de la operación, necesitan del cálculo convencional del resto. Se ha realizado también un análisis para determinar el error del resto calculado de manera alternativa, respecto del convencional, para ver si los resultados obtenidos en las simulaciones eran coherentes. Dichos resultados muestran que el resto calculado de manera convencional solo es necesario en, aproximadamente, el 9 % de los casos para la división y el recíproco. El porcentaje es del 12 % en el caso de la raíz cuadrada y la raíz cuadrada recíproca. Este resultado es incluso mejor que en el caso del método anterior.

5.2. Cálculo alternativo del resto

En esta sección explicamos la manera de obtener el resto de manera alternativa sin utilizar la operación de multiplicación. Como se verá a continuación este método es mucho más sencillo que el explicado en el capítulo anterior. La mejor manera para obtener el resto es hacerlo a partir de una magnitud propia del algoritmo GLD para tener que realizar las mínimas operaciones a a mayores. Empezaremos con el recíproco.

5.2.1. Recíproco

Si nos remontamos a la descripción del algoritmo GLD realizada en la sección 1.2.2, se puede ver que el algoritmo para el recíproco está caracterizado por tres magnitudes que se calculan tal y como se describen a continuación. En primer lugar la magnitud K_i.

$$K_i = 2 - r_{i-1} \tag{5.1}$$

Para obtener K_i es necesario obtener r_i, que se calcula del siguiente modo:

$$r_i = X \cdot K_1 \cdot K_2 \cdots K_i \tag{5.2}$$

Por último, la expresión que calcula el resultado viene dada por:

$$\omega_i = K_1 \cdot K_2 \cdots K_i \tag{5.3}$$

En la implementación de este algoritmo, el cálculo comienza con la obtención de una primera aproximación leída de una tabla ($K_0 = \omega_0$) y en cada iteración se va incorporando una nueva K_i calculada según la ecuación (5.1).

$$\begin{aligned} K_i &= 2 - r_{i-1} \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i \end{aligned} \tag{5.4}$$

Por otra parte, es necesario recordar la expresión convencional del cálculo del resto para el recíproco:

$$rem = 1 - X \cdot \omega_i \tag{5.5}$$

Con todos estos elementos, es necesario fijarse en las ecuaciones (5.2) y (5.3). Como se puede ver, ambas son muy similares y pueden ser relacionadas de la siguiente forma:

$$r_i = X \cdot \omega_i \tag{5.6}$$

Si comparamos la ecuación obtenida con la ecuación (5.5) para el cálculo convencional del resto vemos que son, también muy similares. Sólo se diferencia por la resta de 1. Entonces, debería ser posible obtener un valor para el resto si restamos r_i de 1.

$$rem_i = 1 - r_i \tag{5.7}$$

Una vez que se ha obtenido una expresión alternativa para el cálculo del resto, es necesario realizar algunas consideraciones. En primer lugar, el resto obtenido es el resto del resultado del algoritmo tal y como se obtiene directamente del algoritmo ($\hat{\omega}$). Por lo tanto, al igual que en el método anterior no es posible utilizar el método de redondeo habitual. Se le aplicará el mismo método de redondeo, ya explicado en el apartado 4.4. Por otra parte, también es necesario retomar las consideraciones acerca de la implementación hardware de las operaciones aritméticas. Aquí también ocurre que, al cambiar la secuencia de operaciones para llegar al valor del resto, el resultado puede no ser el mismo en algunas ocasiones. Para ello introducimos más adelante una estimación del error en el cálculo del resto. Es también importante, decir que el proceso a aplicar para el caso de la división sería completamente análogo.

5.2.2. Raíz cuadrada recíproca

En el caso de la raíz cuadrada recíproca hemos de fijarnos en las ecuaciones (1.44), (1.45) y (1.46). En ellas se proponen las distintas magnitudes que in-

tervienen en el cálculo del recíproco de la raíz cuadrada mediante el algoritmo GLD. En primer lugar, K_i.

$$K_i = \frac{3}{2} - \frac{r_{i-1}}{2} \tag{5.8}$$

A continuación r_i.

$$r_i = X \cdot K_1^2 \cdot K_2^2 \cdots K_i^2 \tag{5.9}$$

Y, por último, ω_i

$$\omega_i = K_1 \cdot K_2 \cdots K_i \tag{5.10}$$

En la implementación de este algoritmo, el cálculo comienza con la obtención de una primera aproximación leída de una tabla ($K_0 = \omega_0$) y en cada iteración se va incorporando una nueva K_i calculada según la ecuación (5.8).

$$\begin{aligned} K_i &= \frac{1}{2}(3 - r_{i-1}) \\ \omega_i &= \omega_{i-1} \cdot K_i \\ r_i &= r_{i-1} \cdot K_i^2 \end{aligned} \tag{5.11}$$

La manera convencional de obtener el resto para la raíz cuadrada recíproca es:

$$rem_i = 1 - X \cdot \omega_i^2 \tag{5.12}$$

Una vez que se cuenta con todas las expresiones que entran en juego, es necesario establecer la relación entre las ecuaciones (5.9) y (5.10). También en este caso es muy sencilla.

$$r_i = X \cdot \omega_i^2 \tag{5.13}$$

Por lo tanto, es muy fácil establecer la relación entre la ecuación (5.12) y la ecuación (5.13) que es la misma que para el caso del recíproco.

$$rem_i = 1 - r_i \tag{5.14}$$

Las consideraciones a realizar aquí son las mismas que para el caso del recíproco. Este cálculo del resto solo servirá para establecer un algoritmo de redondeo de latencia variable (por razones ya explicadas). El resultado es fácilmente extensible a la raíz cuadrada. Se presenta a continuación los errores que afectan al cálculo del resto en paralelo.

5.3. Error del resto calculado de manera alternativa

En esta sección se propone la obtención de una estimación del error del resto calculado de manera alternativa. Dicho error se define como la diferencia entre el valor del resto que se obtiene y el valor del resto que se debería obtener. Para ello, como en ocasiones anteriores, para el resto calculado de manera convencional se utilizará la notación $Trem$ y para el resto calculado de manera alternativa se utilizará $Prem$. Se define el error del resto en paralelo (ε_{par}) como:

$$\varepsilon_{alt} = Trem_i - Prem_i \tag{5.15}$$

El siguiente paso es intentar obtener esta expresión en función de distintas contribuciones al error que puedan ser cuantificadas para cada una de las operaciones. Estas contribuciones serán provocadas por la implementación hardware del algoritmo, tal y como se describió en la sección 2.2.

Recíproco: En primer lugar, se obtiene la primera parte de la ecuación (5.13), es decir, $Trem_i$. Se considera, entonces, la implementación hardware de la operación descrita en la ecuación (5.5). Vendrá dada por:

$$Trem_i = 1 - X \cdot \hat{\omega} + \varepsilon^t_{Trem_i} \tag{5.16}$$

Se pueden hacer algunas apreciaciones acerca de esta ecuación. En primer lugar, el resultado $\hat{\omega}$ ahora está afectado por un error con dos componentes. Una debida al error del propio algoritmo y otra debida al error introducido por las operaciones aritméticas.

$$\hat{\omega} = \omega + \varepsilon_i \tag{5.17}$$

No se considera ningún error en los operandos. Por último $\varepsilon^t_{Trem_i}$ representa el error debido, básicamente a la multiplicación y a la resta de 1, necesarios para obtener el resto. Introduciendo la ecuación (5.17) en la ecuación (5.16) se obtiene la expresión final para $Trem_i$.

$$Trem_i = -X \cdot \varepsilon_i + \varepsilon^t_{Trem_i} \tag{5.18}$$

El cálculo en paralelo del resto se obtiene de la implementación en hardware de la ecuación (5.7). El análisis de las distintas contribuciones se hace de manera similar. La expresión del ćcálculo en paralelo del resto es.

$$Prem_i = 1 - \hat{r_i} + \varepsilon^t_{Prem_i} \tag{5.19}$$

Para poder obtener todas las contribuciones al error es necesario explicitar antes las distintas componentes de $\hat{r}_i$.

$$\hat{r_i} = \hat{r}_{i-1} \cdot \hat{K}_i + \varepsilon^t_{r_i} \tag{5.20}$$

Teniendo en cuenta que

$$\hat{r}_{i-1} = X \cdot \hat{\omega}_{i-1} \tag{5.21}$$

se utiliza la expresión anterior transformada de la siguiente forma:

$$\hat{r_i} = (X \cdot \hat{\omega}_{i-1} + \varepsilon^t_{r_{i-1}}) \cdot \hat{K}_i + \varepsilon^t_{r_i} \tag{5.22}$$

Para seguir expandiendo la expresión de $\hat{r}_i$ es necesario introducir las contribuciones que introduce $\hat{K}_i$.

$$\hat{K_i} = 2 - \hat{r}_{i-1} + \varepsilon^t_{K_i} \tag{5.23}$$

Aplicando el mismo razonamiento que en la ecuación (5.22) podemos reescribir la ecuación anterior como:

$$\hat{K_i} = 2 - (X \cdot \hat{\omega}_{i-1} + \varepsilon^t_{r_{i-1}}) + \varepsilon^t_{K_i} \tag{5.24}$$

Introduciendo esta ecuación en la (5.22) y teniendo en cuenta que $\hat{\omega}_{i-1} = \omega + \varepsilon_{i-1}$ se obtiene la expresión final de $\hat{r}_i$. Dicha expresión nos sirve para obtener la expresión del resto calculado en paralelo

$$Prem_i = -X\varepsilon_i - \varepsilon^t_{K_i} - X \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{K_i} - 2\varepsilon^t_{r_{i-1}}) + \varepsilon^t_{r_{i-1}} \cdot (\varepsilon^t_{K_i} - \varepsilon^t_{r_{i-1}}) - \varepsilon^t_{r_i} + \varepsilon^t_{Prem_i} \tag{5.25}$$

No hay más que restar las ecuaciones (5.18) y (5.15) y eliminar un término que es despreciable para obtener la expresión final del resto calculado en paralelo para el recíproco.

$$\varepsilon_{alt} = \varepsilon^t_{K_i} + \varepsilon^t_{r_i} - X \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{K_i} - 2\varepsilon^t_{r_{i-1}}) + \varepsilon^t_{Trem_i} - \varepsilon^t_{Prem_i} \tag{5.26}$$

La expresión obtenida depende de los errores debidos a las distintas operaciones intermedias, necesarias para obtener el resto de manera alternativa. Estas operaciones son precisamente las que provocan las desviaciones del valor del resto respecto de su valor correcto.

Raíz cuadrada recíproca: El proceso para esta operación es muy similar al caso anterior, simplemente hay que adaptar las expresiones específicas para el caso. En primer lugar, el cálculo convencional del resto para la raíz cuadrada recíproca:

$$Trem_i = 1 - X \cdot \hat{\omega}^2 + \varepsilon^t_{Trem_i} \tag{5.27}$$

Siguiendo el mismo proceso que para el recíproco, se puede expandir esta expresión para obtener una expresión para el resto calculado de manera convencional.

$$Trem_i = -X \cdot \varepsilon_i^2 - 2 \cdot \sqrt{X}\varepsilon_i + \varepsilon^t_{Trem_i} \tag{5.28}$$

Por otro lado, para obtener el error del resto, necesitamos obtener la expresión del resto calculado de manera alternativa $Prem_i$.

$$Prem_i = 1 - \hat{r_i} + \varepsilon^t_{Prem_i} \tag{5.29}$$

El primero de los objetivos es la extensión de la magnitud $\hat{r_i}$.

$$\hat{r_i} = \hat{r}_{i-1} \cdot \hat{K}_i^2 + \varepsilon^t_{r_i} \tag{5.30}$$

En la ecuación anterior se hace la siguiente sustitución:

$$\hat{r}_{i-1} = X \cdot \hat{\omega_{i-1}}^2 + \varepsilon^t_{r_{i-1}} \tag{5.31}$$

Solamente hay que explicitar las contribuciones a la magnitud $\hat{K_i}$.

$$\hat{K_i} = \frac{1}{2} \cdot (3 - \hat{r}_{i-1}) + \varepsilon^t_{K_i} \tag{5.32}$$

Siguiendo el mismo esquema que en el caso del recíproco, después de eliminar algunos términos despreciables, se llega a una expresión final para el error del resto calculado en paralelo para la raíz cuadrada recíproca

$$\varepsilon_{alt} = 2 \cdot \varepsilon^t_{K_i} + \varepsilon^t_{r_{i-1}} + \sqrt{X} \cdot \varepsilon_{i-1} \cdot (\varepsilon^t_{i-1} - \varepsilon^t_{K_i} + \varepsilon^t_{K_i^2}) + \varepsilon^t_{Trem_i} - \varepsilon^t_{Prem_i} \tag{5.33}$$

5.4. Validación del método

Una vez construido el modelo teórico para obtener una aproximación del valor del resto de manera alternativa, se realizaron simulaciones para validarlo. En esta sección se presentan los resultados de dichas simulaciones y su comparación con otros métodos. Al igual que en otros métodos presentados estas simulaciones fueron realizadas usando la herramienta de cálculo científico Maple [GLM08]. Como se mencionó anteriormente, este modelo permite proponer un método de redondeo de latencia variable. Por lo tanto, se realizaron diversas simulaciones. En primer lugar, para detectar en que casos el valor del resto obtenido en paralelo no era el adecuado. En segundo lugar, tras proponer el método de latencia variable, se realizaron las simulaciones correspondientes para determinar en que casos había que recurrir al cálculo convencional del resto y en cuales no.

En este caso todas las simulaciones se circunscriben al algoritmo GLD, que es el único para el que es válido este método. El código escrito simula la ejecución hardware del algoritmo incluyendo sus contribuciones al error del resultado. Se realizaron simulaciones tanto para el recíproco como para la raíz cuadrada recíproca. Se incluyó las simulaciones del método de redondeo propuesto y del redondeo tradicional para que puedan ser comparados. Para tener tiempos de simulación razonables se escogió el formato de precisión simple para los resultados. En todos los casos se simuló una iteración del algoritmo, siendo posible obtener datos suficientes sin pérdida de generalidad. La precisión escogida para las operaciones intermedias fue de $n + 5$ bits.

5.4.1. Algoritmo de latencia variable

Como ya se había sugerido en la exposición del modelo teórico, algunos de los valores obtenidos para el resto calculado en paralelo no eran adecuados. Estos producen un valor erróneo al aplicar el método de redondeo descrito en la sección 4.4. Los porcentajes de puntos (respecto del total de puntos posibles) en los que el valor del resto no es adecuado se muestran en la tabla 5.1. Estos porcentajes son muy pequeños (0,9 % para el recíproco y 1,4 % para la raíz cuadrada recíproca) para cualquiera de las operaciones simuladas. En la misma tabla, se presentan también los resultados del método presentado en el capítulo anterior. Son muy similares.

Esta desviación en el valor del resto está provocada por la diferente secuencia de operaciones seguida para el cálculo del resto. Esto hace que los errores debido a la implementación hardware sea distinta y, por lo tanto, el resultado final también. Este resultado tiene como consecuencia que sólo sea posible proponer

	Recip.	Raíz Recip.
GLD (este método)	0.9 %	1.4 %
GLD (resto paralelo)	0.8 %	1.6 %

Tabla 5.1: Porcentaje de casos, respecto del total, donde la acción de redondeo no es correcta

	GLD	
$\hat{\omega}[n+1:n+5]$	Recip.	Raíz Recip.
01110	–	4.7 %
01111	2.1 %	16.2 %
10000	21.6 %	15.9 %
10001	5.7 %	8.6 %

Tabla 5.2: Porcentaje de casos con los mismos últimos 5 bits (respecto del total con la misma terminación) con un valor del resto paralelo incorrecto.

	GLD	
$\hat{\omega}[n+1:n+5]$	Recip.	Raíz Recip.
01101	–	0.8 %
01110	–	9.0 %
01111	2.1 %	21.9 %
10000	17.6 %	14.3 %
10001	4.8 %	5.1 %
10010	–	0.2 %

Tabla 5.3: Obtención del resto en paralelo: Porcentaje de casos con los mismos últimos 5 bits (respecto del total con la misma terminación) con un valor del resto paralelo incorrecto.

un método de latencia variable.

El siguiente paso fue intentar discriminar, mediante simulaciones, para qué valores del resultado ($\hat{\omega}$) se producían valores no válidos del resto paralelo . Dicha clasificación se realizó en función del valor de los 5 últimos bits del resultado, que se calcula con $n+5$ bits de precisión. Se obtuvieron los porcentajes de casos que terminan con los mismos últimos 5 bits, respecto del total de casos con la misma terminación. Estos resultados se pueden ver en la tabla 5.2.

Como se puede ver en dicha tabla, no siempre se calcula un valor adecuado del resto para todos los valores de $\hat{\omega}$ con la misma combinación de los últimos 5 bits. Aunque es solamente un pequeño porcentaje para tres combinaciones determinadas en el caso del recíproco y cuatro combinaciones en el caso de la raíz cuadrada recíproca. Por ejemplo, si los 5 últimos bits de $\hat{\omega}$ son 10000 el cálculo del resto en paralelo es correcto en el 78,4 % de las veces, en el caso del recíproco, y en el 84,1 % de las veces para el caso de la raíz cuadrada recíproca. La aparición de estas combinaciones con valores errores del resto erróneos se producen cuando el error del resto (descrito en la sección anterior) es comparable a la distancia entre el resultado obtenido y el resultado exacto.

Se presentan también los resultados de la misma simulación para el método del capítulo anterior, que obtenía el valor del resto en paralelo (Tabla 5.3). Para el caso del recíproco, los resultados del método propuesto son muy similares a los del cálculo del resto en paralelo. Simplemente hay una ligera diferencia en los porcentajes de casos correspondientes a cada una de las combinaciones de los últimos 5 bits. La diferencia está en el caso de la raíz cuadrada recíproca. En el caso del cálculo en paralelo del resto, aparecen dos combinaciones a mayores que en el nuevo método no aparecen. Esto hace que el método de redondeo que se propone aquí es mas simple. Hay que detectar dos combinaciones menos de los últimos 5 bits menos.

El problema que se presentó aquí es el mismo que para el método anterior. No es posible sintetizar una función lógica o almacenar información en una tabla para predecir los casos concretos que tienen un valor incorrecto del resto paralelo. El área obtenida no sería realizable en un diseño hardware. Por lo tanto, la única posibilidad es calcular el resto de manera convencional cuando se produce una de las combinaciones que aparecen en la tabla 5.2. De este modo, se construye un método de latencia variable.

5.4.2. Eficiencia del método de latencia variable

Se propone, entonces, un método de latencia variable que calcula el valor del resto en paralelo con la ejecución del algoritmo. Si se produce una de las combinaciones de la tabla 5.2, el resto se calcula de manera convencional. Este método tiene un gran ventaja respecto del anterior. La manera de calcular el valor del resto en paralelo que se utiliza ahora apenas consume hardware puesto que la única operación que hay que implementar a mayores es el complemento a uno de la magnitud $\hat{r}$.

Para determinar la calidad del método se realizaron simulaciones para determinar el porcentaje respecto del total en que es necesario el cálculo convencional

del resto. Como es necesario calcular el resto de manera convencional para cada una de las combinaciones problemáticas, es suficiente con contar el número de ocurrencias para cada combinación de los últimos 5 bits para saber cual es el porcentaje respecto del total. Finalmente, la suma de los porcentajes de cada una de las combinaciones que necesitan el cálculo del resto convencional proporciona el porcentaje total. Este es el que mide la eficiencia del método.

	GLD	
$\hat{\omega}[n+1:n+5]$	Recip.	Raíz Recip.
01110	–	3.13 %
01111	3.12 %	3.12 %
10000	3.14 %	3.12 %
10001	3.11 %	3.12 %
Total	9.37 %	12.49 %
Cap. 4	9.37 %	18.74 %
Otros [Sch95]	25.01 %	25.01 %
Otros trabajos[Obe99]	12.49 %	12.49 %

Tabla 5.4: Porcentaje de casos con resto paralelo incorrecto respecto del total

Los resultados de estas simulaciones se encuentran en la tabla 5.4. En la tabla se muestran el porcentaje total de ocasiones en que se necesita el cálculo del resto de manera convencional para cada una de las operaciones. Los porcentajes son de un 9,37 % para el caso del recíproco y de un 12,49 % para la raíz cuadrada recíproca. Se incluyen además, en la parte inferior de la tabla, los resultados del método explicado en el capítulo anterior y en otros trabajos realizados previamente. Como se puede ver, los resultados son mejores en casi todos los casos. Y en los que no el porcentaje es igual, como por ejemplo en el caso del recíproco respecto del método explicado en el método anterior. Aún así el resultado sigue siendo mejor debido a que no es necesario utilizar un multiplicador para obtener el valor del resto. Esto supone un ahorro de hardware muy importante.

5.5. Método propuesto

Esta sección se dedica a la explicación del método propuesto, utilizando los resultados obtenidos de las simulaciones en los apartados anteriores. Es muy similar al propuesto en la sección anterior. El valor del resto que se está obteniendo es el de $(\hat{\omega})$. Por lo tanto es necesario aplicar el método de redondeo explicado

en la sección 4.4. La ventaja ahora es que, para obtener el resto en paralelo no es necesario utilizar la multiplicación. Simplemente hay que complementar la magnitud r_i, de acuerdo con lo explicado en la sección 5.2.

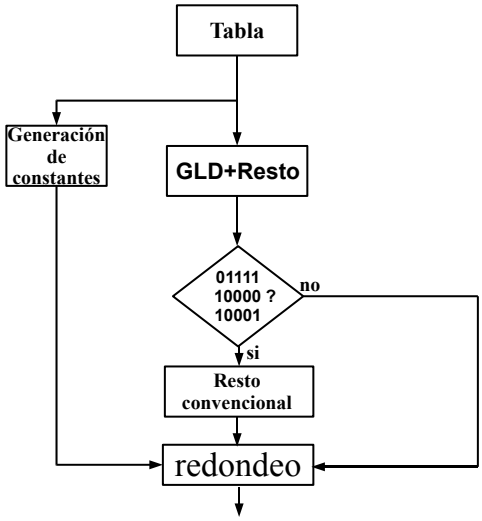
El esquema del método tanto para el recíproco como para la raíz cuadrada recíproca aparece en la figura 4.1. En paralelo con la ejecución del algoritmo, además de la obtención del resto, se generan las constantes necesarias para realizar el redondeo, que sólo dependen del operando. Igual que se había explicado en el método anterior, se almacenan en una tabla de 14 entradas de 28 bits. La constante adecuada se escoge en cuanto se conocen los últimos 5 bits de $\hat{\omega}$. La otra acción en paralelo con la ejecución del algoritmo es el cálculo del resto. En este caso, la implementación es muy sencilla, puesto que no consiste más que en complementar a uno $\hat{r_i}$ de manera que consume muy poca área.

La decisión de si hay que utilizar el resto calculado en paralelo o calcular el resto de manera convencional se toma a través de los últimos 5 bits de $\hat{\omega}$. Si la combinación de los últimos 5 bits coincide con alguna de las de la tabla 5.4. En ese caso será necesario calcular el resto de manera tradicional. Una vez que se ha obtenido el valor del resto se compara con la constante adecuada para determinar la acción de redondeo adecuada.

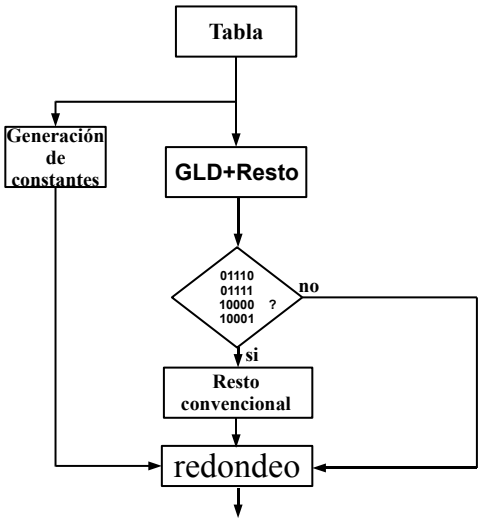
De este modo, es posible utilizar un sólo multiplicador para implementar la ejecución del algoritmo. Si se utiliza un multiplicador segmentado, suponiendo una latencia de 4 ciclos, la latencia para el recíproco es de 5 ciclos por iteración y 9 para la raíz cuadrada recíproca cuando no se utiliza el resto en paralelo. Si se calcula el resto de manera convencional las latencias serán de 9 y 13 ciclos.

5.6. Conclusión

En este capítulo se ha presentado un método que elimina por completo la latencia debida al cálculo del resto para la mayoría de los casos en el redondeo del algoritmo de Goldschmidt. El método se basa en la obtención de un valor del resto en paralelo con la ejecución del algoritmo. El valor del resto obtenido corresponde al resultado del algoritmo $\hat{\omega}$ sin ninguna transformación. Esto hace necesario la utilización de un método de redondeo basado en este valor. La obtención del resto se basa únicamente en el complemento a uno, de manera que apenas consume hardware adicional. Como resultado de la implementación de este método de latencia variable, el cálculo convencional del resto se realiza únicamente en el 9 % para el recíproco y la división. En el caso de la raíz cuadrada y la raíz cuadrada recíproca este porcentaje es del 12 %. Estos resultados han sido obtenidos por medio de simulaciones exhaustivas. Estos resultados



(a) Recíproco



(b) Raíz cuadrada recíproca

Figura 5.1: Esquema propuesto para las unidades con cálculo alternativo del resto

muestran reducciones en el porcentaje de veces que se calcula el resto de manera convencional. Además se produce también una reducción importante en el hardware necesario respecto del método presentado en la sección anterior. Con este método se ahorra un multiplicador, que ahora no se necesita para el cálculo del resto en paralelo con la ejecución del algoritmo.

Conclusiones

La importancia cada vez más grande de la división, el recíproco, la raíz cuadrada y la raíz cuadrada recíproca impone la necesidad de algoritmos para el cálculo de estas funciones que sean más eficientes. Con el paso del tiempo, las soluciones más eficientes se han decantado por el uso de algoritmos multiplicativos debido a su rápido ritmo de convergencia y la posibilidad del reuso del multiplicador ya existente en el sistema.

A lo largo de los últimos años los algoritmos multiplicativos han sido objeto de estudios. La mayoría de ellos han sido dedicados a la mejora de la parte que calcula el resultado. Se consiguieron buenos resultados a través de la obtención de muy diversas formas de obtener buenas aproximaciones iniciales para reducir el número de iteraciones. Por otro lado, también se han realizado avances en la implementación del algoritmo para una obtención más eficiente del resultado. Sin embargo, este tipo de algoritmos tienen un problema que sigue lastrando su rendimiento. No producen de manera directa ni el resultado directamente redondeado ni el resto.

Las soluciones más eficientes adoptan un método de redondeo consistente en la modificación del resultado mediante la suma de una constante y el cálculo del resto de la cantidad del resto. En función del valor del resto y el valor de los bits de redondeo se decide la acción de redondeo adecuado. Este cálculo del resto supone la realización de la multiplicación que en algunos casos puede llegar a ser una penalización del 30 % del tiempo de cálculo total. Todos estos aspectos básicos de introducción al tema han sido descritos en el primer capítulo dedicado a los fundamentos de los algoritmos para el cálculo de la división, la raíz cuadrada y sus recíprocos. También se describe cómo se redondean tradicionalmente estos algoritmos.

Se han propuesto, pues, en este trabajo una serie de métodos para mejorar los métodos de redondeo existentes hasta el momento. Todos ellos son métodos de latencia variable, es decir, métodos que consiguen evitar el cálculo del resto en

un porcentaje de casos respecto del total de casos posibles. Evidentemente, los métodos de latencia variable serán tanto mejores cuanto menor sea el porcentaje de casos en los que es necesario el cálculo del resto. A continuación se presentan las principales contribuciones descritas a lo largo de este trabajo:

En el segundo capítulo se presenta un trabajo previo a la presentación de los distintos métodos de redondeo. Una de los requisitos necesarios para realizar el redondeo de manera adecuada es obtener el resultado con una precisión y una exactitud adecuadas. Se presenta un método de diseño para unidades de división, raíz cuadrada y sus recíprocos pare el algoritmo de Goldschmidt (el mismo método para el algoritmo de Newton-Raphson se presenta en un apéndice al final de este trabajo). Este método permite determinar el tamaño óptimo de las operaciones intermedias para un tamaño de aproximación inicial y un número de iteraciones determinado a partir de los requerimientos de precisión y exactitud.

Este método está basado en un análisis preciso del error que tiene en cuenta las distintas contribuciones al error del resultado del algoritmo para una iteración dada. De este análisis se obtiene una expresión analítica del error final del resultado de una iteración cualquiera. A través de esta expresión es posible calcular límites para el valor de dicho error. A través de estas expresiones, fijando el tamaño de la aproximación inicial se obtiene el tamaño de palabra óptimo para las operaciones intermedias.

Obtener un tamaño de palabra óptimo reduce entre otras cosas el tamaño del multiplicador produciendo un ahorro de área y de tiempo en las implementaciones hardware. Como resultado, este método fue aplicado al rediseño de la unidad del AMD-K7. Se obtuvieron mejoras significativas en forma de reducción del número de productos parciales del multiplicador, es decir, de su tamaño. Tanto con respecto a la propia solución como a otros métodos previos que ya mejoraban dicha implementación.

El tercer capítulo presenta el primero de los métodos de latencia variable que mejora la eficiencia de los métodos preexistentes. Este método es una modificación de los métodos previos y es independiente del algoritmo multiplicativo utilizado. Basta con obtener el resultado con la exactitud adecuada. Permite una reducción de los casos donde el cálculo del resto es necesario en un 40 %. El cálculo del resto se produce en un 15 % de las ocasiones cuando se usa un bit extra en la aproximación al resultado a redondear. Este porcentaje es de un 9 % si se usan dos bits en dicha aproximación. Porcentajes aún menores se consiguen si se aumentan el número de bits extra de la aproximación.

Este método está basado en utilizar información adicional para evitar el cálculo del resto en más casos que en el método tradicional. En el redondeo

convencional, el resultado obtenido del algoritmo se modifica para obtener otra aproximación con algunos bits extra, además de aquellos requeridos por la precisión mínima necesaria. Teniendo en cuenta el valor de estos bits extra se decide si es necesario o no realizar el cálculo del resto. En este nuevo método se examinan, también, bits del resultado obtenido directamente del algoritmo. Examinando estos bits, además de los usados tradicionalmente, se reduce el número de casos en los que es necesario el cálculo del resto.

El cuarto capítulo contiene la descripción de un nuevo método de redondeo. Este forma parte de un grupo de dos métodos, que se presentan en este capítulo y el siguiente, que son completamente nuevos. El resultado de la aplicación de este método es que se consigue evitar la latencia asociada al cálculo del resto en una gran parte de los casos. Concretamente, al 9 % de los casos para la división y el recíproco, y al 15 % en el caso de la raíz cuadrada y la raíz cuadrada recíproca. Estamos, entonces ante un método de latencia variable.

El método se compone de la obtención del valor del resto en paralelo con la ejecución del algoritmo y un nuevo método de redondeo. Se propuso un método que obtiene el resto del resultado directamente obtenido del algoritmo. El cálculo del resto implica la operación de multiplicación. Esto obligó al diseño de un método basado en esta aproximación al resultado sin modificar. Consiste en comparar el resto con constantes fácilmente calculables. La constante utilizada depende de los últimos bits del resultado obtenido. Los resultados obtenidos muestran importantes reducciones en el número de veces en que hay que calcular el resto de manera tradicional respecto de trabajos previos.

El último de los capítulos presenta también un método para el algoritmo de Goldschmidt que obtiene un valor para el resto en paralelo con su ejecución. Evita el cálculo convencional del resto en la mayoría de los casos. Dicho cálculo se realiza en el 9 % de las veces para la división y el recíproco. 12 % en el caso de la raíz cuadrada y la raíz cuadrada recíproca. Igual que en el caso anterior, el valor del resto obtenido corresponde al resultado del algoritmo sin ninguna modificación. La diferencia es que el cálculo del resto es mucho más sencillo que en el caso anterior. De esta manera se ahorra la necesidad de un multiplicador adicional. Esto provoca un ahorro de hardware muy significativo en las implementaciones respecto del método anterior.

Apéndice A

Análisis del error para el algoritmo de Newton-Raphson

En el capítulo 2 se presentó un método de diseño de unidades funcionales para el cálculo de la división, el recíproco, la raíz cuadrada y la raíz cuadrada recíproca en punto flotante. Aunque se mencionó que este método podía ser aplicado a cualquier algoritmo multiplicativo, por simplicidad se realizó únicamente el análisis para el caso del algoritmo de Goldschmdit (GLD). Este apéndice completa este análisis y presenta el análisis del error y el cálculo de los límites de este para el algoritmo de Newton-Raphson (NR), que es el otro algoritmo multiplicativo más representativo. Al igual que para el algoritmo GLD se realizarán los análisis para el recíproco y la raíz cuadrada recíproca. Estos resultados pueden ser trivialmente extendidos para la división y la raíz cuadrada. Este apéndice se divide en tres partes. En primer lugar, la obtención del análisis del error para el algoritmo NR. De este análisis se extraerán los límites del error. Por último se describirá la validación de los resultados obtenidos por medio de simulación.

A.1. Introducción

El análisis explicado a continuación es muy similar al ya explicado para el algoritmo GLD. También son válidas aquí todas aquellas consideraciones ge-

nerales que se hicieron acerca de la estructura de las implementaciones de los algoritmos multiplicativos y de la notación. Se considerará un esquema que parte de una aproximación inicial de tamaño b y que realizará dos iteraciones. En el análisis del error se contabilizarán las contribuciones de las operaciones intermedias (de tamaño p) que se acumula al error de aproximación del propio algoritmo. El objetivo del análisis del error es, también, obtener una expresión para el error de la aproximación al resultado en una iteración general i. Estas expresiones serán utilizadas para obtener los valores máximos y mínimos para el error en cada iteración. Estos valores serán utilizados para obtener el tamaño óptimo de las operaciones intermedias. Se presentarán únicamente los análisis para el recíproco y la raíz cuadrada recíproca, puesto que los resultados para división y raíz cuadrada son fácilmente extraíbles a partir de éste.

A.2. Análisis del error

A.2.1. Recíproco

La obtención del análisis del error absoluto del resultado se basa en ir incorporando al algoritmo (1.27) las distintas contribuciones al error. Las ecuaciones del algoritmo NR para el recíproco son las siguientes:

$$\begin{aligned} n_{i-1} &= X \cdot \omega_{i-1} \\ d_{i-1} &= 2 - n_{i-1} \\ \omega_i &= \omega_{i-1} \cdot d_{i-1} \end{aligned} \tag{A.1}$$

Se denomina como iteración 0 a la acción de obtener la aproximación inicial al resultado. Teniendo en cuenta que el tamaño de esta aproximación es b, su error viene dado por:

$$\varepsilon_0 = 2^{-b} \tag{A.2}$$

A partir de la primera ecuación (1.27) para obtener la magnitud n, se obtiene

$$\hat{n}_0 = X \cdot \hat{\omega_0} + \varepsilon_{n_0}^t \tag{A.3}$$

Esta ecuación incorpora, además del error de aproximación, el error debido a la multiplicación necesaria para calcular $\hat{n_0}$. Esta expresión puede ser expandida para obtener,

$$\hat{n}_0 = X \cdot (\omega + \varepsilon_0) + \varepsilon_{n_0}^t \tag{A.4}$$

La siguiente ecuación, utilizada para obtener la magnitud d, se puede reescribir de la siguiente forma.

$$\hat{d}_0 = 2 - \hat{n}_0 + \varepsilon_{d_0}^t \tag{A.5}$$

Combinando esta ecuación con la (A.4) se obtiene.

$$\hat{d}_0 = 2 - X \cdot (\omega + \varepsilon_0) - \varepsilon_{n_0}^t + \varepsilon_{d_0}^t \tag{A.6}$$

Finalmente, la expresión que proporciona el resultado es:

$$\hat{\omega}_1 = \hat{\omega}_0 \cdot \hat{d}_0 + \varepsilon_{\omega_1}^t \tag{A.7}$$

Incluyendo los resultados de (A.6) se obtiene la expresión final del resultado para la primera iteración con todas las contribuciones al error.

$$\hat{\omega_1} = (\omega + \varepsilon_0) \cdot [2 - X \cdot (\omega + \varepsilon_0) - \varepsilon_{n_0}^t + \varepsilon_{d_0}^t] + \varepsilon_{\omega_1}^t \tag{A.8}$$

Teniendo en cuenta que $\hat{\omega}_1 = \omega + \varepsilon_1$, el error de la aproximación al resultado para la primera iteración (ε_1) es:

$$\varepsilon_1 = -X \cdot \varepsilon_0^2 - \omega \cdot (\varepsilon_{n_0}^t - \varepsilon_{d_0}^t) - \varepsilon_0 \cdot (\varepsilon_{n_0}^t - \varepsilon_{d_0}^t) + \varepsilon_{\omega_1}^t \tag{A.9}$$

Se puede seguir el mismo procedimiento para obtener una expresión del error para una iteración general i. Para ello partimos de la ecuación de la magnitud n para una iteración $i - 1$.

$$\hat{n}_{i-1} = X \cdot \hat{\omega}_{i-1} + \varepsilon_{n_{i-1}}^t \tag{A.10}$$

Igual que en el caso particular anterior, esta ecuación puede ser expandida.

$$\hat{n}_{i-1} = X \cdot (\omega + \varepsilon_{i-1}) + \varepsilon_{n_{i-1}}^t \tag{A.11}$$

La siguiente ecuación que hay que abordar es la que sirve para calcular d.

$$\hat{d}_{i-1} = 2 - \hat{n}_{i-1} + \varepsilon_{d_{i-1}}^t \tag{A.12}$$

Esta ecuación se expande como sigue:

$$\hat{d}_{i-1} = 2 - X \cdot (\omega + \varepsilon_{i-1}) - \varepsilon_{n_{i-1}}^t + \varepsilon_{d_{i-1}}^t \tag{A.13}$$

El resultado de la operación de la iteración i lo proporciona la siguiente ecuación:

$$\hat{\omega}_i = \hat{\omega}_{i-1} \cdot \hat{d}_{i-1} + \varepsilon^t_{\omega_i} \tag{A.14}$$

Incluyendo la expresión obtenida en (A.12) se obtiene el resultado con todas las contribuciones al error.

$$\hat{\omega}_i = (\omega + \varepsilon_{i-1}) \cdot [2 - X \cdot (\omega + \varepsilon_{i-1}) - \varepsilon^t_{n_{i-1}} + \varepsilon^t_{d_{i-1}}] + \varepsilon^t_{\omega_i} \tag{A.15}$$

Lo que finalmente nos sirve para obtener la expresión final del error de la aproximación al resultado para una iteración cualquiera i.

$$\varepsilon_i = -\frac{1}{\omega} \cdot \varepsilon^2_{i-1} - \omega \cdot (\varepsilon^t_{n_{i-1}} - \varepsilon^t_{d_{i-1}}) - \varepsilon_{i-1} \cdot (\varepsilon^t_{n_{i-1}} - \varepsilon^t_{d_{i-1}}) + \varepsilon^t_{\omega_i} \tag{A.16}$$

La estructura del error es muy parecida al del algoritmo (GLD). Está compuesta por el error de aproximación, que es el primer término de la ecuación anterior, y los errores de las operaciones de los demás términos, representados por el resto de términos. Dado que no se considera ningún error en los operandos, para obtener el error para la división, no habría más que multiplicar la expresión anterior por el dividendo Y.

A.2.2. Raíz cuadrada recíproca

El análisis para la raíz cuadrada recíproca es similar al anterior.
Las ecuaciones del algoritmo NR para la raíz cuadrada recíproca son:

$$\begin{aligned} s_{i-1} &= \omega^2_{i-1} \\ n_{i-1} &= X \cdot n_{i-1} \\ d_{i-1} &= 3 - n_{i-1} \\ \omega_i &= \frac{\omega_{i-1}}{2} \cdot d_{i-1} \end{aligned} \tag{A.17}$$

Se parte, igualmente, de una aproximación inicial de tamaño b. El error de esta aproximación es:

$$\varepsilon_0 = 2^{-b} \tag{A.18}$$

El algoritmo NR para la raíz cuadrada recíproca está descrito en (1.27). Incluyendo el error debido a la operación de multiplicación en la primera de las ecuaciones, obtenemos:

$$\hat{s}_0 = \hat{\omega}_0^2 + \varepsilon_{s_0}^t \tag{A.19}$$

La expansión de la ecuación anterior produce el siguiente resultado

$$\hat{s}_0 = \omega^2 + \varepsilon_0^2 + 2 \cdot \omega \cdot \varepsilon_0 + \varepsilon_{s_0}^t \tag{A.20}$$

El siguiente paso es el cálculo de la magnitud n.

$$\hat{n}_0 = X \cdot \hat{s}_0 + \varepsilon_{n_0}^t \tag{A.21}$$

Utilizando la expresión para $\hat{s}_0$ obtenida en A.20, se obtiene:

$$\hat{n}_0 = 1 + \frac{1}{\omega^2} \cdot \varepsilon_0^2 + \frac{2}{\omega} \cdot \varepsilon_0 + \frac{1}{\omega^2} \cdot \varepsilon_{s_0}^t + \varepsilon_{n_0}^t \tag{A.22}$$

$\hat{n}_0$ se utiliza para obtener $\hat{d}_0$.

$$\hat{d}_0 = \frac{1}{2} \cdot (3 - \hat{n}_0) + \varepsilon_{d_0}^t \tag{A.23}$$

La expresión final de $\hat{d}_0$ es:

$$\hat{d}_0 = 1 - \frac{1}{2 \cdot \omega^2} \cdot \varepsilon_0^2 - \frac{1}{\omega} \cdot \varepsilon_0 - \frac{1}{2 \cdot \omega^2} \varepsilon_{s_0}^t - \frac{1}{2} \cdot \varepsilon_{n_0}^t + \varepsilon_{d_0}^t \tag{A.24}$$

La magnitud que nos da el resultado para la primera iteración es la siguiente:

$$\hat{\omega}_1 = \hat{\omega}_0 \cdot \hat{d}_0 + \varepsilon_{\omega_1}^t \tag{A.25}$$

Introduciendo el valor de $\hat{d}_0$ en la ecuación anterior se obtiene el resultado para la primera iteración con todas las contribuciones al resultado.

$$\hat{\omega}_1 = (\omega + \varepsilon_0) \cdot (1 - \frac{1}{2 \cdot \omega^2} \cdot \varepsilon_0^2 - \frac{1}{\omega} \cdot \varepsilon_0 - \frac{1}{2 \cdot \omega^2} \varepsilon_{s_0}^t - \frac{1}{2} \cdot \varepsilon_{n_0}^t + \varepsilon_{d_0}^t) + \varepsilon_{\omega_1}^t \tag{A.26}$$

A partir de la ecuación del resultado se obtiene el error para la primera iteración:

$$\varepsilon_1 = -\frac{3}{2\omega}\cdot\varepsilon_0^2 - \frac{1}{2\omega^2}\cdot\varepsilon_0^3 - \frac{1}{2\omega}\varepsilon_{s_0}^t - \frac{1}{2\omega^2}\cdot\varepsilon_0\cdot\varepsilon_{s_0}^t$$
$$-\omega\cdot(\frac{\varepsilon_{n_0}^t}{2}+\varepsilon_{d_0}^t) - \varepsilon_0\cdot(\frac{\varepsilon_{n_0}^t}{2}+\varepsilon_{d_0}^t) + \varepsilon_{\omega_1}^t \quad \text{(A.27)}$$

Siguiendo los mismos pasos que en el caso del recíproco, es posible obtener la expresión del error de al aproximación al resultado para una iteración genérica i.

$$\varepsilon_i = -\frac{3}{2\omega}\cdot\varepsilon_{i-1}^2 - \frac{1}{2\omega^2}\cdot\varepsilon_{i-1}^3 - \frac{1}{2\omega}\varepsilon_{s_{i-1}}^t - \frac{1}{2\omega^2}\cdot\varepsilon_{i-1}\cdot\varepsilon_{s_{i-1}}^t$$
$$-\omega\cdot(\frac{\varepsilon_{n_{i-1}}^t}{2}+\varepsilon_{d_{i-1}}^t) - \varepsilon_{i-1}\cdot(\frac{\varepsilon_{n_{i-1}}^t}{2}+\varepsilon_{d_{i-1}}^t) + \varepsilon_{\omega_i}^t \quad \text{(A.28)}$$

A.3. Límites del error

El análisis del error anterior se usa, en este apartado, para obtener las expresiones de los valores máximo y mínimo del error de la aproximación al resultado para dos iteraciones del algoritmo. Se utilizarán las mismas suposiciones que las ya hechas para el algoritmo GLD. Estos valores máximo y mínimo estarán dados en función del tamaño de la aproximación inicial (b) y del tamaño de las operaciones intermedias (p).

A.3.1. Recíproco

Se calcularán los límites para la primera y la segunda iteración del algoritmo. Se obtienen los límites para el recíproco. La obtención de los mismos para la división es trivial. Sólo hay que multiplicar los límites obtenidos por el valor máximo del dividendo, que es 2.

Una iteración: Se parte de la ecuación del error de la aproximación al resultado para la primera iteración.

$$\varepsilon_1 = -\frac{1}{\omega}\cdot\varepsilon_0^2 - \omega\cdot(\varepsilon_{n_0}^t - \varepsilon_{d_0}^t) - \varepsilon_0\cdot(\varepsilon_{n_0}^t - \varepsilon_{d_0}^t) + \varepsilon_{\omega_1}^t \quad \text{(A.29)}$$

De acuerdo con las consideraciones, ya hechas, acerca del error de cada una de las contribuciones al error se obtiene que:

$$\begin{array}{llll} -2^{-b} & \leq \varepsilon_0 \leq & 2^{-b} \\ -2^{-p-1} & \leq \varepsilon_{n_0}^t \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon_{d_0}^t \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon_{\omega_1}^t \leq & 2^{-p-1} \end{array} \tag{A.30}$$

De este modo, se puede calcular el límite de cada uno de los términos de la ecuación A.29. Los límites para el primer término serán:

$$-2^{-2\cdot b+1} \leq -\frac{1}{\omega} \cdot \varepsilon_0^2 \leq 0 \tag{A.31}$$

Este término nunca tiene un valor positivo. Los límites para el siguiente de los términos son:

$$-2^{-p} \leq \omega \cdot (\varepsilon_{n_0}^t - \varepsilon_{d_0}^t) \leq 2^{-p} \tag{A.32}$$

Los del tercer término son:

$$-2^{-b} \cdot 2^{-p} \leq \varepsilon_0 \cdot (\varepsilon_{n_0}^t - \varepsilon_{d_0}^t) \leq 2^{-b} \cdot 2^{-p} \tag{A.33}$$

El término que queda es el error de la multiplicación necesaria para obtener el resultado de la primera iteración ($\hat{\omega}_1$). Tiene los límites:

$$-2^{-p-1} \leq \varepsilon_{\omega_1}^t \leq 2^{-p-1} \tag{A.34}$$

Mediante la agrupación de los límites superiores e inferiores, respectivamente se obtienen los límites globales para la primera iteración.

$$-2^{-2b+1} - 2^{-p-1} \cdot (3 + 2^{-b+1}) \leq \varepsilon_1 \leq 2^{-p-1} \cdot (3 + 2^{-b+1}) \tag{A.35}$$

Dos iteraciones: En este caso se parte de la expresión del error para la segunda iteración:

$$\varepsilon_2 = -\frac{1}{\omega} \cdot \varepsilon_1^2 - \omega \cdot (\varepsilon_{n_1}^t - \varepsilon_{d_1}^t) - \varepsilon_1 \cdot (\varepsilon_{n_1}^t - \varepsilon_{d_1}^t) + \varepsilon_{\omega_2}^t \tag{A.36}$$

En las distintas contribuciones al error hay que tener en cuenta, ahora, los límites de la aproximación al resultado para la primera iteración obtenidos anteriormente. De este modo, las distintas contribuciones al error son las siguientes:

$$\begin{array}{rcl} -2^{-2b+1} - 2^{-p-1} \cdot (3 + 2^{-b+1}) & \leq \varepsilon_1 \leq & 2^{-p-1} \cdot (3 + 2^{-b+1}) \\ -2^{-p-1} & \leq \varepsilon_{n_1}^t \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon_{d_1}^t \leq & 2^{-p-1} \\ -2^{-p-1} & \leq \varepsilon_{\omega_2}^t \leq & 2^{-p-1} \end{array} \tag{A.37}$$

Al igual que en el caso anterior, el siguiente paso es la obtención de los límites para cada uno de los términos de la expresión del error. Entonces para el primero de los términos los límites son:

$$\begin{array}{rcl} -\dfrac{1}{\omega}\varepsilon_1^2 & \geq & -2 \cdot [2^{-2b+1} + 2^{-p-1} \cdot (3 + 2^{-b+1})]^2 \\ -\dfrac{1}{\omega}\varepsilon_1^2 & \leq & 0 \end{array} \tag{A.38}$$

Se pueden aplicar las mismas aproximaciones que fueron aplicadas a la ecuación (2.38) en el capítulo 2. Así, los límites para el primer término serán:

$$\begin{array}{rcl} -\dfrac{1}{\omega}\varepsilon_1^2 & \leq & 0 \\ -\dfrac{1}{\omega}\varepsilon_1^2 & \geq & -2^{-4b+3} - 3 \cdot 2^{-2b+2} \cdot 2^{-p} \end{array} \tag{A.39}$$

Los límites para el segundo término se pueden extraer fácilmente:

$$-2^{-p} \leq \omega \cdot (\varepsilon_{n_1}^t - \varepsilon_{d_1}^t) \leq 2^{-p} \tag{A.40}$$

Haciendo las aproximaciones adecuadas se obtiene la expresión final para los límites del tercer término:

$$-2^{-p} \cdot 2^{-2b+1} \leq \varepsilon_1 \cdot (\varepsilon_{n_1}^t - \varepsilon_{d_1}^t) \leq 2^{-p} \cdot 2^{-2b+1} \tag{A.41}$$

Los límites del último término son iguales que en la iteración anterior.

$$-2^{-p-1} \leq \varepsilon_{\omega_2}^t \leq 2^{-p-1} \tag{A.42}$$

La agrupación de todos los límites de cada uno de los términos produce la expresión del límite del error para la segunda iteración del recíproco:

$$\begin{array}{rcl} \varepsilon_2 & \leq & 2^{-p-1} \cdot (3 + 2^{-2b+2}) \\ \varepsilon_2 & \geq & -2^{-4b+3} - 2^{-p-1} \cdot (3 + 7 \cdot 2^{-2b+2}) \end{array} \tag{A.43}$$

A.3.2. Raíz cuadrada recíproca

Se presenta, a continuación, la obtención de los límites del error de la aproximación al resultado para la raíz cuadrada recíproca. Igual que en el caso anterior, se proponen los límites para una y dos iteraciones.

Una iteración: La expresión del error para la primera iteración es la siguiente:

$$\begin{aligned}\varepsilon_1 &= \frac{2}{2\omega}\cdot\varepsilon_0^2 - \frac{1}{2\omega^2}\cdot\varepsilon_0^3 - \frac{1}{2\omega^2}\varepsilon_{s_0}^t - \frac{1}{2\omega^2}\varepsilon_0\varepsilon_{s_0}^t \\ &\quad -\omega\cdot(\frac{\varepsilon_{n_0}^t}{2} - \varepsilon_{d_0}^t) - \varepsilon_0\cdot(\frac{\varepsilon_{n_0}^t}{2} - \varepsilon_{d_0}^t) + \varepsilon_{\omega_1}^t \end{aligned} \tag{A.44}$$

El siguiente paso es ver los límites del error término a término. El primero de ellos tiene el siguiente límite.

$$-3\cdot 2^{-2\cdot b} \leq -\frac{3}{2\omega}\varepsilon_0^2 \leq 0 \tag{A.45}$$

A continuación, el segundo término.

$$-2^{-3\cdot b+1} \leq -\frac{1}{2\omega^2}\cdot\varepsilon_0^3 \leq 2^{-3\cdot b+1} \tag{A.46}$$

Estos son los términos que corresponden al error de aproximación del algoritmo. Los siguientes incluyen errores debidos a las operaciones intermedias. El tercer término tiene los siguientes límites:

$$-2^{-p} \leq -\frac{1}{2\omega^2}\varepsilon_{s_0}^t \leq 2^{-p} \tag{A.47}$$

Los del cuarto término son:

$$-2^{-b}\cdot 2^{-p} \leq -\frac{1}{2\omega^2}\cdot\varepsilon_0\cdot\varepsilon_{s_0}^t \leq 2^{-b}\cdot 2^{-p} \tag{A.48}$$

El quinto tiene los límites:

$$-\frac{3}{2} \leq 2^{-p-1} \leq \omega\cdot(\frac{\varepsilon_{n_0}^t}{2} - \varepsilon_{d_0}^t) \leq \frac{3}{2} \leq 2^{-p-1} \tag{A.49}$$

A continuación, el sexto término.

$$-\frac{3}{2} \leq 2^{-p-1}\cdot 2^{-b} \leq \varepsilon_0\cdot(\frac{\varepsilon_{n_0}^t}{2} - \varepsilon_{d_0}^t) \leq \frac{3}{2} \leq 2^{-p-1}\cdot 2^{-b} \tag{A.50}$$

Finalmente, el último término.

$$-2^{-p-1} \leq \varepsilon_{\omega_1}^t \leq 2^{-p-1} \tag{A.51}$$

Para obtener la expresión final de los límites del error, se agrupan los distintos términos y se aplica, al límite inferior, la aproximación ya descrita en la gráfica 2.2. Se obtienen los siguientes límites:

$$\begin{aligned} \varepsilon_1 &\leq 2^{-3b+1} + \frac{9}{2} \cdot 2^{-p-1} + \frac{7}{2} \cdot 2^{-p-1} \cdot 2^{-b} \\ \varepsilon_1 &\geq -3 \cdot 2^{-2b} - \frac{9}{2} \cdot 2^{-p-1} - \frac{7}{2} \cdot 2^{-p-1} \cdot 2^{-b} \end{aligned} \tag{A.52}$$

Dos iteraciones: El punto de partida es la expresión del error para la segunda iteración:

$$\begin{aligned} \varepsilon_2 = \; & \frac{2}{2\omega} \cdot \varepsilon_1^2 - \frac{1}{2\omega^2} \cdot \varepsilon_1^3 - \frac{1}{2\omega^2} \varepsilon_{s_1}^t - \frac{1}{2\omega^2} \varepsilon_1 \cdot \varepsilon_{s_1}^t \\ & -\omega \cdot (\frac{\varepsilon_{n_1}^t}{2} - \varepsilon_{d_1}^t) - \varepsilon_1 \cdot (\frac{\varepsilon_{n_1}^t}{2} - \varepsilon_{d_1}^t) + \varepsilon_{\omega_2}^t \end{aligned} \tag{A.53}$$

Para el primer término hay que tener en cuenta la expresión de los límites de la iteración anterior (A.52). Además se desprecian los términos que son múltiplos de 2^{-2p} y se aplica la aproximación descrita en la figura 2.3. De este modo, los límites de error correspondiente al primer término son:

$$-27 \cdot 2^{-4b} + 81 \cdot 2^{-2b} \cdot 2^{-p-1} \leq \frac{-3}{2\omega} \varepsilon_1^2 \leq 0 \tag{A.54}$$

En el caso del segundo término, las aproximaciones necesarios son la eliminación de los términos que son múltiplos de 2^{-2p} y la aproximación descrita en la gráfica 2.4. Los límites que se obtienen son:

$$-27 \cdot 2^{-6b+1} + 243 \cdot 2^{-4b} \cdot 2^{-p-1} \leq \frac{-1}{2\omega^2} \cdot \varepsilon_1^3 \leq 27 \cdot 2^{-6b+1} + 243 \cdot 2^{-4b} \cdot 2^{-p-1} \tag{A.55}$$

Los límites del tercer término son más sencillos de obtener:

$$-2^{-p} \leq \frac{1}{2\omega^2} \varepsilon_{s_1}^t \leq 2^{-p} \tag{A.56}$$

Para el cuarto término, despreciando los términos que son múltiplos de 2^{-2p}, se obtienen los siguientes límites:

$$-6 \cdot 2^{-2 \cdot b} \cdot 2^{-p-1} \leq -\frac{1}{2\omega^2}\varepsilon_1 \cdot \varepsilon_{s_1}^t \leq 6 \cdot 2^{-2 \cdot b} \cdot 2^{-p-1} \tag{A.57}$$

Los límites del error del quinto término son:

$$-\frac{3}{2} \cdot 2^{-p-1} \leq -\omega \cdot (\frac{\varepsilon_{n_1}^t}{2} - \varepsilon_{d_1}^t) \leq \frac{3}{2} \cdot 2^{-p-1} \tag{A.58}$$

En el caso del sexto término se desprecian aquellos términos que son múltiplos de 2^{-2p}:

$$-\frac{9}{2} \cdot 2^{-2 \cdot b} \cdot 2^{-p-1} \leq \varepsilon_1 \cdot (\frac{\varepsilon_{n_1}^t}{2} - \varepsilon_{d_1}^t) \leq \frac{9}{2} \cdot 2^{-2 \cdot b} \cdot 2^{-p-1} \tag{A.59}$$

El último término tiene los límites:

$$-2^{-p-1} \leq \varepsilon_{\omega_2}^t \leq 2^{-p-1} \tag{A.60}$$

Para obtener la expresión de los límites del error hay que hacer varias aproximaciones. En primer lugar, se despreciaran, igual que en los casos anteriores, todos los términos que son múltiplos de 2^{-2p}. Además de esto, se aplicará la aproximación descrita en la gráfica 2.5 para los límites superiores y las aproximaciones del gráfico 2.6 para los límites inferiores.

Finalmente, la expresión obtenida para los límites del error de la aproximación al resultado en la segunda iteración son los siguientes.

$$2^{-4b+5} + \frac{9}{2} \cdot 2^{-p-1} \leq \varepsilon_2 \leq 2^{-6b+6} + \frac{9}{2} \cdot 2^{-p-1} \tag{A.61}$$

A.4. Simulaciones

Para el caso del algoritmo NR también se realizaron simulaciones para la validación de las expresiones analíticas del error obtenidas y los valores máximo y mínimo del error. Del mismo modo que para el algoritmo GLD, se realizaron simulaciones exhaustivas, se simuló el formato del estándar IEEE754 para precisión simple ($b = 7$, $p = 26$) y se utilizó un esquema de dos iteraciones para poder validar todas la expresiones obtenidas.

A.4.1. Recíproco

El primer aspecto que se comprobó fue si, realmente, el algoritmo simulado producía los resultados adecuados. Es decir, que se da una convergencia correcta del algoritmo.

Las figuras A.1a, A.1b y A.1c se representan los resultados de las simulaciones para cada una de las iteraciones. En la primera de ellas se puede ver la aproximación inicial al resultado. Ya, a partir de la primera iteración, se observa la rápida convergencia del resultado hacia el resultado exacto..

El siguiente aspecto que era necesario comprobar era si las expresiones analíticas obtenidas para el error del resultado reproducían su comportamiento real. Dicho error era obtenido mediante su medición durante la simulación de la ejecución del algoritmo. Los casos relevantes, son la primera y la segunda iteración. Se comprueba además si los valores máximos y mínimos obtenidos para el error son adecuados.

Las figuras A.2a y A.2b se puede ver el error del resultado para la primera iteración. El error medido y el obtenido de las ecuaciones del modelo coinciden de manera exacta. Se puede ver también como los valores límite obtenidos son acordes con el error obtenido.

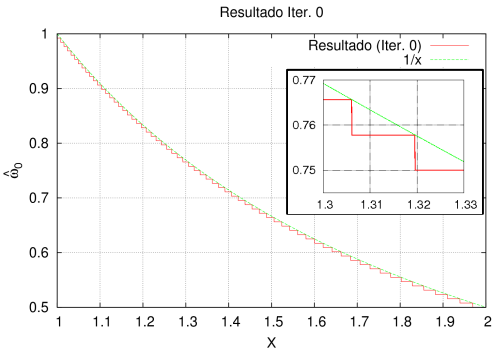
Las figuras A.3a y A.3b son las gráficas correspondientes al error del resultado para la segunda iteración. El error medido y el error calculado de manera teórica coinciden. También se incluye en ambas gráficas los valores de los límites para la segunda iteración. Se puede ver que estos límites predicen el rango de valores que puede tomar el error del resultado para la segunda iteración.

En las figuras A.4a, A.4b y A.4c representa el error de cada una de las tres iteraciones en escala logarítmica.

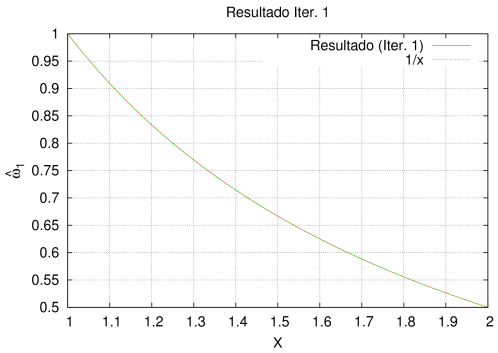
A.4.2. Raíz cuadrada recíproca

También se realizaron las simulaciones para la raíz cuadrada recíproca. Los parámetros utilizados de estas simulaciones son los mismos que en todos los casos anteriores. En esta sección se presentan los resultados de dichas simulaciones. La primera de las comprobaciones consiste en asegurarse de que el resultado converge adecuadamente hacia el resultado exacto. La aproximación inicial y el resultado para las dos iteraciones se presentan en las gráficas. Se aprecia como el algoritmo converge rápidamente hacia el resultado deseado.

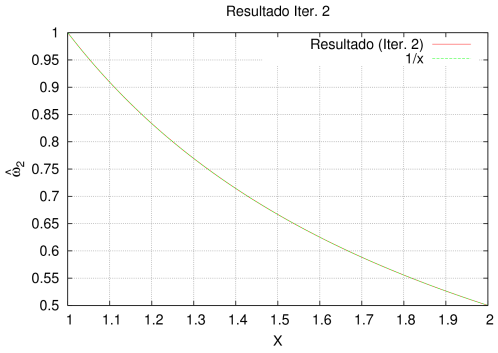
Una vez comprobado que el resultado es correcto, es necesario validar el modelo teórico para el error. Se compara el error medido de cada iteración con el error obtenido del modelo teórico evaluando la ecuación (A.28). Se comprueba



(a) Resultado del recíproco para la Iteración 0

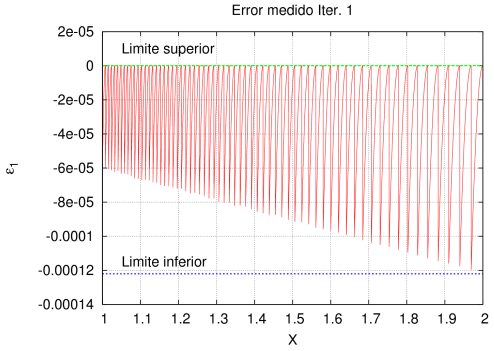


(b) Resultado del recíproco para la Iteración 1

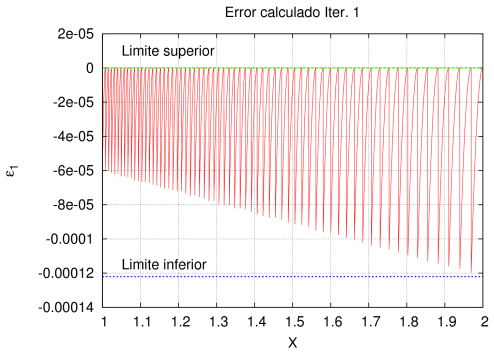


(c) Resultado del recíproco para la Iteración 2

Figura A.1: Error del recíproco en las distintas iteraciones para el algoritmo NR

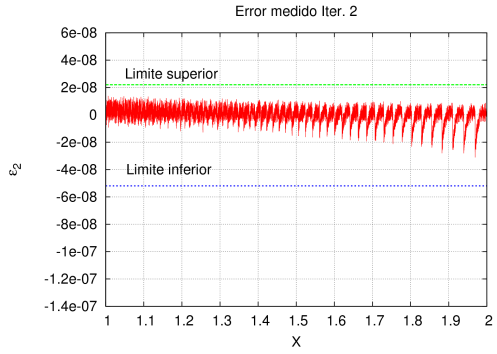


(a) Error medido en la primera iteración para el recíproco

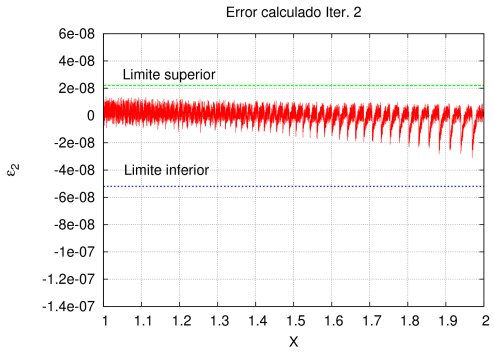


(b) Error obtenido de las expresiones teóricas en la primera iteración para el recíproco

Figura A.2: Error del resultado del recíproco para la primera iteración del algoritmo NR

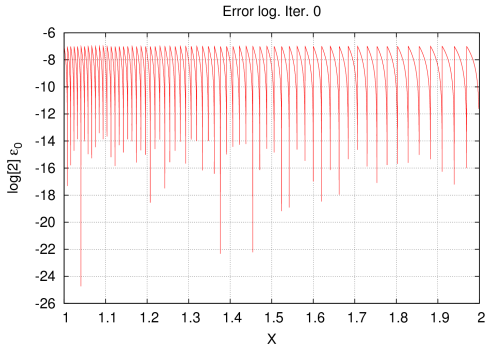


(a) Error medido en la segunda iteración para el recíproco

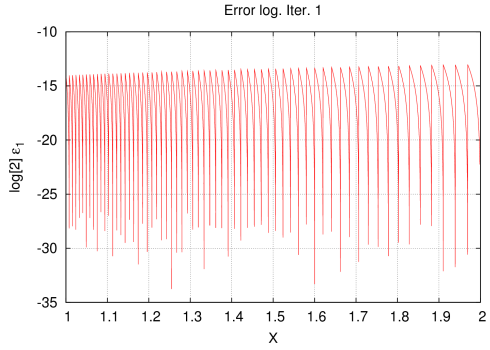


(b) Error obtenido de las expresiones teóricas en la segunda iteración para el recíproco

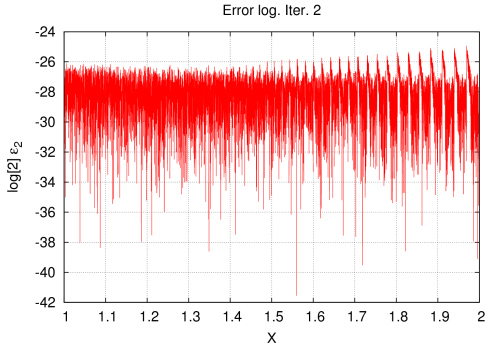
Figura A.3: Error del resultado del recíproco para la segunda iteración del algoritmo NR



(a) Error logarítmico para la Iteración 0 del recíproco

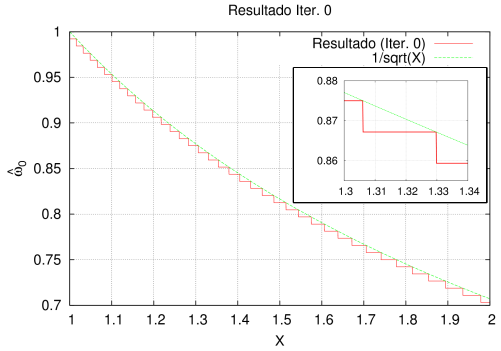


(b) Error logarítmico para la Iteración 1 del recíproco

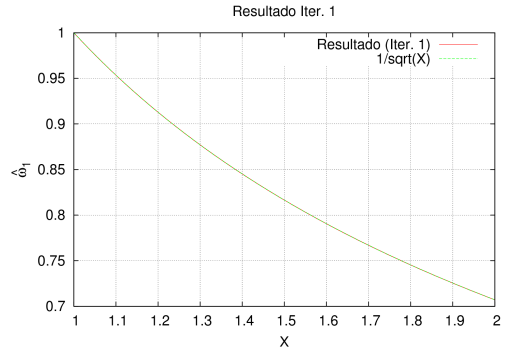


(c) Error logarítmico para la Iteración 2 del recíproco

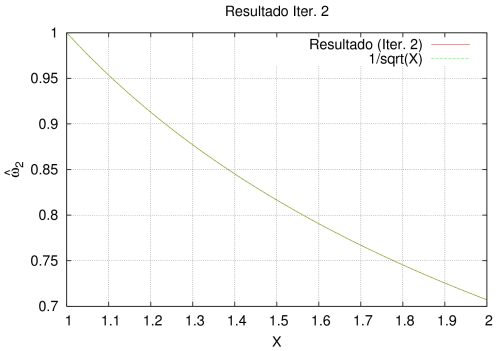
Figura A.4: Error logarítmico del recíproco en las distintas iteraciones para el algoritmo NR



(a) Resultado de la raíz cuadrada recíproca para la Iteración 0



(b) Resultado de la raíz cuadrada recíproca para la Iteración 1



(c) Resultado de la raíz cuadrada recíproca para la Iteración 2

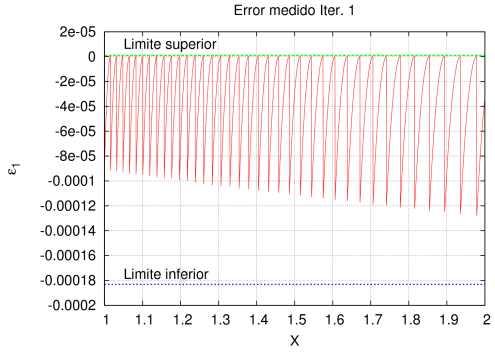
Figura A.5: Error de la raíz cuadrada recíproca en las distintas iteraciones para el algoritmo NR

también la validez de los límites obtenidos para el error. Al igual que en todos los casos anteriores el caso simulado es una longitud de las operaciones intermedias de $p = 26$ y un tamaño de aproximación inicial de $b = 7$.

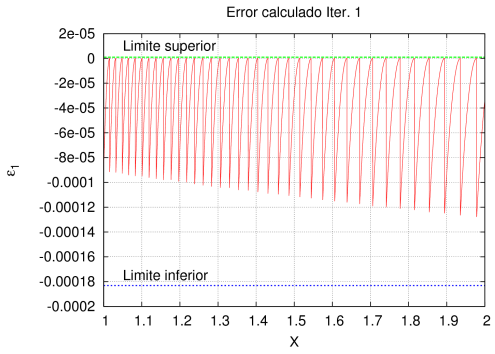
En las figuras A.6a and A.6b se presenta las gráficas con los resultados de las simulaciones para la primera iteración. Como se puede ver, tanto el error medido como el obtenido del modelo coinciden exactamente. Además los límites del error son adecuados

Las mismas simulaciones proporcionaron también los resultados para la segunda iteración. Los resultados se muestran en las figuras A.7a y A.7b. El modelo teórico reproduce de manera adecuada el mismo comportamiento que muestra el error medido. Debido a los efectos del error introducido por las operaciones intermedias, se puede ver como el número de puntos con error positivo aumenta considerablemente con respecto a la primera iteración.

Del mismo modo que en las operaciones anteriores se presentan el error en escala logarítmica para conocer que bits fraccionales están afectados por el error.

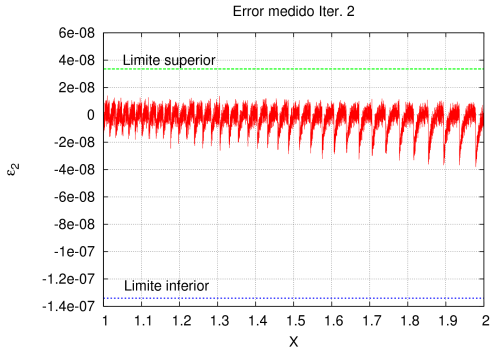


(a) Error medido en la primera iteración para la raíz cuadrada recíproca

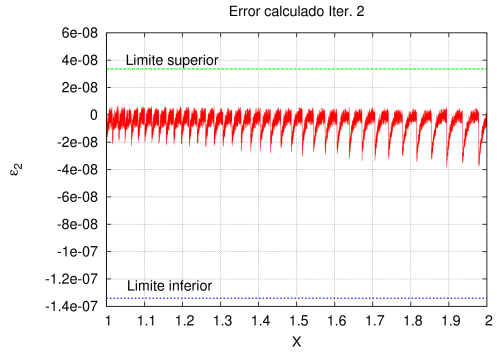


(b) Error obtenido de las expresiones teóricas en la primera iteración para la raíz cuadrada recíproca

Figura A.6: Error del resultado de la raíz cuadrada recíproca para la primera iteración del algoritmo NR

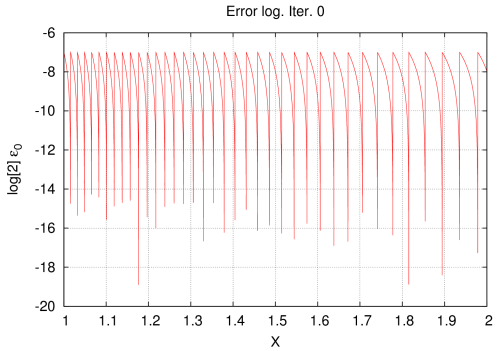


(a) Error medido en la segunda iteración para la raíz cuadrada recíproca

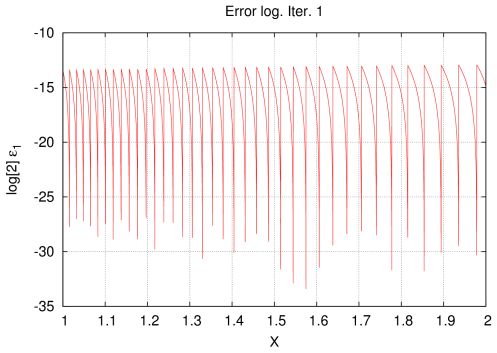


(b) Error obtenido de las expresiones teóricas en la segunda iteración para la raíz cuadrada recíproca

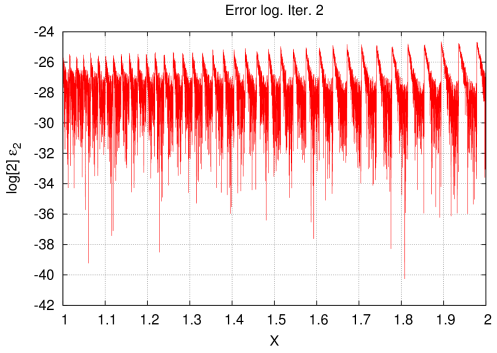
Figura A.7: Error del resultado del recíproco para la segunda iteración del algoritmo NR



(a) Error logarítmico para la Iteración 0 del recíproco



(b) Error logarítmico para la Iteración 1 del recíproco



(c) Error logarítmico para la Iteración 2 del recíproco

Figura A.8: Error logarítmico del recíproco en las distintas iteraciones para el algoritmo NR

Bibliografía

[Bla06] N. Blachford. *Inside the Physx Physics processor*, 2006.

[CL94] J. Cortadella and T. Lang. High-radix division and square root with speculation. *IEEE Transactions on Computers*, 43(8), 1994.

[CS05] D. Chatterje and M. Sachdev. Design of a 1.7-ghz low-power delay-fault-testable 32-b alu in 180-nm cmos technology. *IEEE transactions on very large scale of integration*, 13(13), November 2005.

[DHH97] S. Oberman D. Harris and M. Horowitz. SRT division architectures and implementations. In *Proceedings of the IEEE 13th. Intl. Symposium on Computer Arithmetic (ARITH13)*, pages 18–25, 1997.

[EL89] M. D. Ercegovac and T. Lang. Fast radix-2 division with quotient-digit prediction. *Journal of VLSI Signal Processing*, 2(1):169–180, 1989.

[EL92] M. D. Ercegovac and T. Lang. On-the-fly rounding. *IEEE Transactions on Computers*, 41(12), 1992.

[EL94a] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Kluwer Academic Publishers, 1994.

[EL94b] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, 1994.

[ES00] G. Even and P. M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. *IEEE Transactions on Computers*, 49(7), 2000.

[ES03a] G. Even and P. -M. Seidel. A parametric error analysis of Goldschmidt's division algorithm. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic(ARITH'03)*, 2003.

[ES03b] G. Even and P. M. Seidel. Pipelined multiplicative division with ieee rounding. In *Proceedings of the 21st International Conference on Computer Design (ICCD'03)*. IEEE Computer Society, 2003.

[Fan87] J. Fandrianto. Algorithm for high speed shared radix-4 division and radix-4 square root. In *Proceedings of the 8th IEEE Symposium on Computer Arithmetic (ARITH8)*, 1987.

[Fer67] D. Ferrari. A division method using a parallel multiplier. *IEEE Transactions on Computers*, EC-16, 1967.

[FO01] M. J. Flynn and S. F. Oberman. *Advanced Computer Arithmetic Design*. Wiley-Blackwell, 2001.

[GLM08] K. Geddes, G. Labahn, and M. Monagan. *Maple 12 Advanced Programming Guide*. Maplesoft, 2008.

[Gol64] R. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, June 1964.

[Gol91] D. Goldbert. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991.

[GWSH33] G. Gerwig, H. Wetter, E. M. Scwarz, and J. Haess. High performance floating point unit with 116 bit wide divider. In *Proceedings of the 16th IEEE Symposium of Computer Arithmetic (ARITH'03)*. IEEE Computer Society, 2033.

[IEE08] IEEE Computer Society. *IEEE standard for floating-point arithmetic*, 2008. IEEE Std 754-2008.

[IHE+00] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, and A. Kunimatsu. 2.44-gflops 300 mhz floating point vector processing unit for high-performance 3d graphics computing. *IEEE Journal of Solid-State Circuits*, 35(7), July 2000.

[IM99] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH14)*, 1999.

[ITY95] M. Ito, N. Takagi, and S. Yajima. Efficient initial approximation and fast converging methods for division and square root. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH12)*, 1995.

[LA01] T. Lang and E. Antelo. Correctly rounded reciprocal square root by digit-recurrence and radix-4 implementation. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH15)*, 2001.

[Mar90] P. Markstein. Computation of elementary function on the IBM RISC system/6000 processor. *IBM J. Research and Development*, pages 111–119, January 1990.

[Mat01] D. W. Matula. Improved table lookup algorithms for postscaled division. In *Proceedings of the 15th Symposium on Computer Arithmetic (ARITH 15)*. IEEE Computer Society, 2001.

[NDJD01] A. Naini, A. Dhablania, W. James, and D. DasSarma. 1-ghz hal sparc64 dual floating point unit with ras features. In *Proceedings of the 15th Symposium on Computer Arithmetic (ARITH 15)*. IEEE Computer Society, 2001.

[NL98] A. Nannarelli and T. Lang. Low-power radix-8 divider. In *Proceedings of International Conference on Computer Design (ICCD)*, 1998.

[NL99a] A. Nannarelli and T. Lang. Low-power divider. *IEEE Transactions on Computers*, 48(1):2–14, January 1999.

[NL99b] A. Nannarelli and T. Lang. Low-power division: Comparison among implementations of radix 4, 8 16. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH14)*, 1999.

[Obe99] S. F. Oberman. Floating point division and square root implementation in the AMD-K7 microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (ARITH14)*, 1999.

[OF96] S. F. Oberman and M. J. Flynn. Implementing division and other floating point operations: A system perspective. *Scientific Computing and Validated Numerics*, pages 18–24, 1996.

[OF97] S.F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, February 1997.

[Par99] K. K. Parhi. *VLSI Digital Signal Processing Systems. Design and Implementation.* Wiley-Interscience, 1999.

[PB02] J. A. Piñeiro and J. D. Bruguera. High-speed double precision computation of reciprocal, division, square root and inverse square root. *IEEE Transactions on Computers*, 51(13), 2002.

[PB03] J. A. Piñeiro and J. D. Bruguera. On-line high-radix exponential with selection by rounding. In *Proceedings of the 2003 International Symposium on Circuits and Systems*, 2003.

[PEB03] J. A. Piñeiro, M. D. Ercegovac, and J. D. Bruguera. High-radix logarithm with selection by rounding: Algorithm and implementation. *Journal of VLSI*, 2003.

[PZ95] J. A. Prahbu and G. B. Zyner. 167 mhz radix-8 divide and square root usign overlapped radix-2 stages. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH12)*, 1995.

[Sch95] E. Schwarz. Rounding quadratically converging algorithms for division and square root. In *Proceedings of the 29th Asilomar Conference on Signals and Systems*, 1995.

[Sco85] N. R. Scott. *Computer Number Systems and Arithmetic.* Prentice Hall, 1985.

[Tak01] N. Takagi. A hardware algorithm for computing reciprocal square root. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH15)*, 2001.

[TF96] G. B. Thomas and R. L. Finney. *Calculus and Analytic Geometry.* Addison Wesley, 1996.

[WF82] S. Waser and M. J. Flynn. *Introduction to Arithmetic to Digital Systems Designers.* CBS College Publishing, 1982.